Distributed Parallel Extreme Event Analysis in Next Generation Simulation Architectures

by

Stephen S. Hamilton

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2017

Abstract

Numerical simulations present challenges as they reach exascale because they generate petabyte-scale data that cannot be saved without interrupting the simulation due to I/O constraints. Data scientists must be able to reduce, extract, and visualize the data while the simulation is running, which is essential for in transit and post analysis. Next generation architectures in supercomputing include a burst buffer technology composed of SSDs primarily for the use of checkpointing the simulation in case a restart is required. In the case of turbulence simulations, this checkpoint provides an opportunity to perform analysis on the data without interrupting the simulation.

First, we present a method of extracting velocity data in high vorticity regions. This method requires calculating the vorticity of the entire dataset and identifying regions where the threshold is above a specified value. Next we create a 3D stencil from values above the threshold and dilate the stencil. Finally we use the stencil to extract velocity data from the original dataset. The result is a dataset that is over an order of magnitude smaller and contains all the data required to study extreme

events and visualization of vorticity.

The next extraction utilizes the zfp lossy compressor to compress the entire velocity dataset. The compressed representation results in a dataset an order of magnitude smaller than the raw simulation data. This provides the researcher approximate data not captured by the velocity extraction. The error introduced is bounded, and results in a dataset that is visually indistinguishable from the original dataset.

Finally we present a modular distributed parallel extraction system. This system allows a data scientist to run the previously mentioned extraction algorithms in a distributed parallel cluster of burst buffer nodes. The extraction algorithms are built as modules for the system and run in parallel on burst buffer nodes. A feature extraction coordinator synchronizes the simulation with the extraction process. A data scientist only needs to write one module that performs the extraction or visualization on a single subset of data and the system will execute that module at scale on burst buffers, managing all the communication, synchronization, and parallelism required to perform the analysis.

Primary Reader: Randal Burns

Secondary Reader: Charles Meneveau

Tertiary Reader: Alexander S. Szalay

Acknowledgments

First I would like to thank Dr. Randal Burns for his timely strategic advice and support during my entire Ph.D process. He consistently kept me on track and helped me avoid losing time chasing ideas that were not productive while still providing me the freedom to choose my own work. His timely encouragement during the research process was essential for keeping me on track, and I am truly grateful for it.

I would like to thank Dr. Charles Meneveau for helping me understand the necessary parts of Turbulence science to conduct my research. He also provided excellent clear guidance on improving the Johns Hopkins Turbulence Databases. He has inspired me to learn about the fluid dynamics field of Mechanical Engineering.

I would also like to thank Dr. Alexander S. Szalay for teaching me how to expand my knowledge outside computer science by exposing me to such concepts as the Golomb ruler. He also always ensured I had all the resources necessary to conduct my work, and constantly challenged our practices to improve the JHTDB.

I thank my collaborators, Dr. Peter Lindstrom, Perry Johnson, and Dr. Kalin Kanov for helping me understand aspects of science that were necessary to complete

this dissertation.

I also thank my lab partners, Kunal Lillaney, Disa Mhembere, Alex Baden, Da Zheng, and James Browne. They provided technical advice when I needed it, and also great comradarie that was essential during the late nights.

I must also thank Paul Stanton for introducing me to Randal.

Finally, I would like to thank my wife, Danielle, and my two sons Jason and Aaron for supporting me throughout my entire Ph.D. process.

This work is supported in part by the National Science Foundation under Grants CMMI-0941530, OCI-108849, ACI-1261715, No. OCI-1244820, CBET-1507469, and AST-0939767, Johns Hopkins University's Institute for Data Intensive Engineering & Science, Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and under the auspices of the U.S. Department of Energy.

Dedication

This dissertation is dedicated to my father, John B. Hamilton. He always encouraged me to continue my higher education, and was the first person I can remember to call me Dr. Stephen as a child.

Contents

CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

Supercomputing trends toward exascale present the problem of an increasing performance gap between processing and I/O. At exascale, simulations will output fewer than one byte for every 10^5 bytes of system state; they will produce 200-300 PB/s in memory[1] and only 1 TB/s will be saved to persistent storage.[2] Going forward, the lack of I/O bandwidth to long term storage will slow down the simulation by an order of magnitude. The I/O required for checkpointing simulations to file systems has become the performance limiting workload in scalable HPC[3] and exposure to failure governs checkpoint frequency.

In order to address this problem, there has been a recent architecture change to the memory hierarchy of high performance computing (HPC) clusters. This change is the addition of burst buffers which are slower than main memory but faster than hard drives. The Trinity supercomputer at Los Alamos deployed a burst buffer ar-

chitecture[4] to fill the performance gap between cluster memory and disk filesystems. Burst buffers place the SSD storage on the fast network to catch I/O bursts that would overwhelm the filesystem. Data on burst buffers are short lived; they must be discarded or stored to file system in (tens of) minutes.

While this architecture directly addresses the performance gap between processing and I/O for checkpoint storage, it does not solve the problem of long term storage of simulation data. Therefore, these next generation architectures must define meaningful ways to output data that preserve scientific discovery on reduced data representations.

We develop methods that capture and extract relevant scientific data of a direct numerical simulation as it runs. Our experiences using the Johns Hopkins Turbulence Databases[5] (JHTDB) inform the choice of data products that we extract from burst buffers. We propose a model in which checkpoints are written to burst buffers at the frequency needed for analysis. Then we extract a subset of the data and reduced representations that can be utilized for scientific analysis and visualization in real-time as well as post simulation. The extraction requires little processing power and it does not disrupt the running simulation. On Trinity,[6] the burst buffers are located on additional nodes that are separate from compute nodes. Burst buffers in recent architectures collocate compute and SSDs[7] and extraction codes can be run within the burst buffer nodes. Specifically, we extract high-resolution velocity data from regions of relatively high vorticity, and store a lower resolution dataset that is compressed

with the zfp lossy compressor that is error bounded.

1.1 Related Work

1.1.1 Feature Extraction Methods

Supercomputing continues to evolve with speed increases and hardware architecture changes that coincide with application development to leverage these new architectures. Bent et al.[8] explore burst buffer configurations and demonstrate that placing SSDs between compute nodes and the storage array allow jitter-free co-processing of their visualization tasks and reduce total time to completion by up to thirty percent. We utilize a similar architecture in our work. Ma et al.[9] discuss in-situ data extraction and visualization. They modify the simulation code to provide data useful for visualization in-situ, whereas our work performs feature extraction in-transit via burst buffers without having to modify existing simulation codes. Ahrens et al.[10] describe and test methods of utilizing multi-core CPU and GPU based processors in the Roadrunner supercomputer to perform visualization of an exascale simulation in-situ. Chen et al.[11] utilize the HemeLB lattice-Boltzmann code for large-scale fluid flow. They discuss pre- and post-processing along with computational steering to modify simulation parameters in situ. This work differs from ours in the way the data is saved and utilized for post-processing. They create a multi-resolution data structure by storing their simulation output in a hierarchical order. This method allows for

visualization without reading the entire dataset. Motivated by an interest to avoid losing intense events that may be sparsely distributed in space and may be absent in low-resolution representations, in our work, we utilize the SSD burst buffers to read the entire timestep and perform thresholding and extraction of high-magnitude events on a per-timestep basis. Wang et al.[12] developed a file system (BurstFS) that aggregates I/O bandwidth from burst buffers and maintains a distributed key-value store of metadata for the files. This system allows an application to perform small non-contiguous read operations on the burst buffer. Because our feature extraction reads of all the data, this file system would not benefit our work.

We build upon the concept of burst buffers[13] to integrate non-volatile memory into the supercomputing storage hierarchy. We focus specifically on using the SSD to capture write bursts, particularly those from checkpoint workloads. Other concept papers have discussed using burst buffers more generally in the HPC memory hierarchy.[14]

1.1.2 Distributed Parallel Extraction

Our system which we call Myrcene performs distributed an parallel extraction on burst buffer nodes. Blanas et al. present a system called Scientific Data Services (SDS/Q)[15] which provides a query interface for the Hierarchical Data Format version 5 (HDF5) that runs in a parallel distributed environment. This can operate on any type of scientific data stored in the HDF5 format and abstracts the parallel and

distributed operation. However, the SDS/Q system differs from Myrcene in that it was designed for querying as opposed to computationally-intensive feature extraction and visualization. The goal of the SDS/Q system was to create a system that would outperform relational database systems. Tournavitis and Franke[16] present a semi-automatic method of compiling applications written as single thread into a multicore parallel application on a single node. This is a very generic approach since it can work on any code, however it does not address the distributed architecture of a cluster.

The automated parallelization of functions in Myrcene was inspired in part by Map/Reduce,[17] which has been implemented and extended by many parallel systems such as Hadoop[18] and Spark.[19] Myrcene inherits the notions of data-parallel execution and functional parallelism. Unlike Map/Reduce which performs sequential I/O, Myrcene uses the burst-buffer SSDs to support arbitrary data access patterns. Fast I/O for SSDs have been used as a building block for graph-processing[20] and linear algebra[21] systems. These are again limited to parallelism within a single node. Pearce et al.[22] demonstrates a distributed implementation of graph-analysis on SSDs. This is a specific implementation of graph traversal algorithms and not a general execution framework.

1.2 Johns Hopkins Turbulence Databases

The majority of this work utilizes datasets from the JHTDB. The JHTDB contains multiple datasets from direct numerical simulations of the Navier-Stokes equations representing turbulent fluid flow that range from tens to 150 terabytes. In particular, the isotropic turbulence dataset contains 5028 timesteps of velocity with three components of floating point values and one component of floating point pressure values on a 1024^3 spatially dense regular grid. This dataset provides scientists all over the world an opportunity to discover many aspects of turbulence without the need to run their own large simulation. A number of discoveries from the JHTDB have come from the combination of visualization and analysis of high vorticity regions. These include a vorticity hierarchy that is not evident on smaller scale simulations[23] and that magnetic flux freezing in high-conductivity plasmas fails in the presence of MHD turbulence, explaining why solar flares can erupt in minutes or hours rather than the millions of years predicted by flux freezing.[24]

1.3 Velocity Extraction from High Vorticity Regions

In the first part of the extraction process we use the isotropic turbulence dataset from the JHTDB to extract a subset of velocity data in 3D space only at points where

the vorticity magnitude exceeds a defined threshold.[1] Next we dilate the volumes within this 3D space by a kernel size based on the requirements for post analysis and extract the velocity field in the dilated regions. The dilation allows us to capture data just outside the high vorticity regions needed for iso-surface extraction and Lagrangian interpolation in post-processing. Many filters and derivative equations also rely on this additional data gained from the dilation for interpolation kernels around the region, which makes the extracted data useful for scientific analysis. The result is a sparse dataset on a 3D structured grid that is an order of magnitude or more smaller than the original data. The actual size of the extracted data is directly impacted by the threshold chosen prior to extraction.

1.4 Lossy Compression with zfp

Velocity extraction from high vorticity regions deliberately leaves out regions of low vorticity. Understanding that we cannot save the entire dataset, we extract a separate dataset that contains full field lower precision data by using lossy compression. We leverage the zfp algorithm,[26] which is specifically designed to compress floating point scientific data in 1D, 2D, or 3D space. zfp's lossy compression is *error-bounded*; it guarantees that the values differ from the original by less than a specified amount. zfp achieves an order of magnitude or more compression and the loss of accuracy is

[1] Thresholds are easy to choose because turbulence has threshold values with physical meaning derived from the inverse Kolmogorov scale. Multiples of this scale may be used to describe the near absence of, medium, and high vorticity[25]

indistinguishable when visualizing the data. These characteristics lend themselves well to capturing exascale simulation data for visualization not only for turbulence data but for any scientific data in a structured grid.

1.5 Myrcene

In order to utilize the velocity extraction and zfp compression in a massively parallel and distributed burst buffer node cluster, we present a system called Myrcene. We named the system after the myrcene essential oil in plants that is extracted for its aromatic and flavor qualities due to the fact that we are performing extraction of data essential for scientific discovery. Myrcene simplifies the process by allowing the data scientist to focus on the algorithm and only parameters required to process a small subset of the data in a standalone module. Once the module is written, the scientist configures the computation through the Myrcene web interface, specifying the number of nodes available for computation, the compute resources (cores) on those nodes, the shape of the data, and filenames and locations of data. Using this information, a feature extraction coordinator will direct the nodes to perform the extraction. A small modular client runs on the burst buffer nodes that await signaling from the feature extraction coordinator to perform requested tasks. During execution, metadata about the extraction computation time is collected and updated in realtime. Myrcene stores this information to allow scientists to view node performance during

the extraction process.

In addition to simplifying the distributed and parallel programming effort, the system also provides built-in reporting functionality and storage of run time metadata. Typically these data are not gathered since the scientist is focused on the simulation outputs, however the data could prove to provide further insight into simulations if differences occur. For example, if extraction takes more time on certain parts of a dataset the automatic reports would reveal this immediately allowing the data scientist to take action if necessary. The timing issue could potentially be anything from simulation hardware issues, coding issues, or a calculation that takes longer on a particular part of the dataset. Since compute time is expensive, this provides yet another data capture point that can help the scientist understand the details of the extraction after the simulation completes without requiring additional runs.

Chapter 2

Velocity Extraction from High Vorticity Regions

The first extraction method we created generates a dataset in which velocities are stored in regions of relatively high vorticity, and regions of low vorticity contain zero values. In addition to storing the velocity in high vorticity regions, we also include neighboring points to the high vorticity regions. These points are included to allow future contouring and interpolation since these functions require these neighboring values. There are multiple steps to generate this dataset. We first use a vorticity calculation method (Q-criterion or Vorticity Magnitude) to generate a vorticity value at each point. Next we use a threshold and create a 3D stencil where points above the threshold are set to one, and all other points set to zero. Then we dilate this stencil by a specified kernel size. Finally we use the stencil to "cut" out velocity values from

the original dataset resulting in a dataset with values outside the stencil turning to zero. The new dataset is approximately an order of magnitude smaller but varies depending on the chosen threshold.

2.1 Identifying Vorticity

In turbulent flows, identification of coherent structures, specifically vortices, aids in scientific understanding of these flows. Inside and around these high vortical regions, energy dissipation and squared vorticity (enstrophy) are orders of magnitude higher than the mean values, which we refer to as extreme events.[27] There are various methods for identifying vortices. Vortices are defined by the velocity field that reflects the rotational qualities and there is not a single approved method to describe vortices. Dubief and Delcayre[28] examine four methods of vortex identification: pressure, vorticity magnitude, λ_2, and Q-criterion. Because pressure fails to capture fine details in isotropic turbulence[28] and λ_2 appears to be affected by small noise present in all data, we examine visualizations based on vorticity magnitude and Q-criterion. Each of these two methods provide good visualizations of vortical flow structure when utilized to generate iso-surface visualizations. However, one particular issue with vorticity magnitude is that the vorticity criterion does not distinguish between swirling motions and shearing motions. Thus, vorticity magnitude can also present layered structures that are vorticity sheets and not vortices.[29] Q-criterion is also not perfect

as it fails to reliably identify Bödewadt vorticies. However note that such vorticies occur normal to a wall, and the isotropic turbulence dataset used in our analysis is periodic and does not contain any walls.[30]

2.2 Vorticity Magnitude vs Q-criterion

Since Q-criterion and vorticity magnitude appear to be acceptable for visualization, we examine properties of each. First we analyze the performance of the vorticity magnitude and Q-criterion for generating vortical flow iso-surfaces on an isotropic turbulence dataset. In order to compare Q-criterion versus vorticity performance we defined a threshold that is equivalent for each calculation. The thresholds and resulting data can be constrained based on either scientific concerns (the loss of accuracy when evaluating averages of gradient norms over the entire flow volume) or system resources that set a target data size. This adjustment allows us to produce data that fits within available storage in the computing center, while still gathering useful scientific data to study these high vorticity regions. In order to determine the threshold, we begin by using a multiple of the root-mean-square value of the vorticity fluctuations. This value is known a-priori, based on knowledge of the dissipation rate ϵ and fluid viscosity ν according to $\langle\vec{\omega}\cdot\vec{\omega}\rangle^{1/2} = \sqrt{\epsilon/\nu}$[31] where ω is the vorticity vector (curl of the velocity). For the data from the JHTDB, this value is $\sqrt{.0928/.000185} = 22.4$, which is also the inverse Kolomogorov time scale τ_η. Since we are interested in high vorticity

regions, we scale this low reference threshold to achieve a clear visual representation of high vorticity regions.

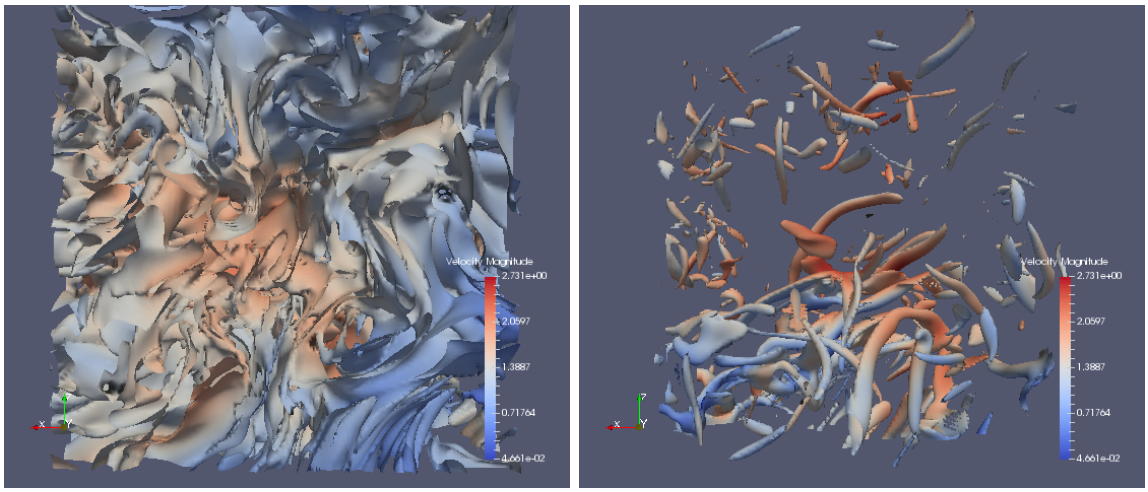

Figure 2.1: Vorticity magnitude contour at threshold 22.4 (left) and 55.98 (right)

We tested various multiples of $1/\tau_\eta$ and found that a multiple of 2.5 presented clear vorticity structures without obvious erroneous surfaces. The threshold chosen in this case is $2.5 * 22.4 = 55.98$. The visualization of vorticity magnitude at this threshold was a much clearer representation of vortices than using a threshold of 22.4 as seen in Figure 2.1.

Upon finding a reasonable threshold, we calculated the equivalent threshold for Q-criterion. In the absence of straining motions, the relationship between the threshold of vorticity and Q-criterion can be taken to be as follows: $Q = \frac{1}{4}\omega^2$. Therefore the threshold value for Q that we chose is $Q = .25(55.98)^2 = 783$.

Figure 2.2 displays the visualization of vorticity magnitude contour versus the Q-criterion contour. Though they look very similar, the bottom left corner of the

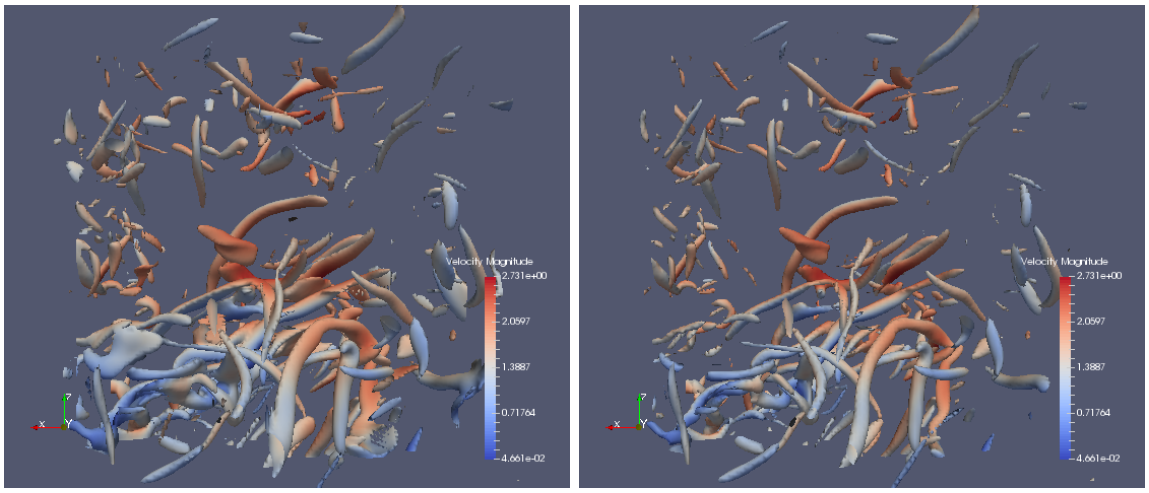

Figure 2.2: Vorticity magnitude (left) and Q-criterion (right)

left image (vorticity magnitude) displays a structure that is not present in the Q-criterion visualization. This is due to shearing, because the vorticity magnitude does not differentiate between shearing and curl. In the definition of Q, strain is subtracted from vorticity which results in a lower Q value and filters out shearing. We performed

Cube Size	Vorticity Threshold	Q Threshold
64	.257	.222
128	1.597	1.375
192	5.070	4.500
256	11.120	9.522

Table 2.1: Vorticity vs. Q Thresholding in seconds total time per cube on a single core

additional tests at various thresholds and cube dimensions (subsets of the full 1024^3 grid in the JHTDB) to determine whether the computation of Q-criterion or vorticity magnitude has an impact on overall feature extraction time. Table 2.1 compares total computation times, which includes reading from and writing to the burst buffer. Our

results show that Q-criterion computes slightly faster than vorticity regardless of cube size.

2.2.1 Iso-surface Representation of Vorticies

The next metric we tested for vorticity extraction was data reduction of Q-criterion versus vorticity magnitude. As mentioned previously, the iso-surface generation from the vorticity computation results in triangles that form a mesh of vortical structures. The only data required for this representation are the points in space for each triangle which results in a significant reduction of data from the original dense velocity values. In Figure 2.2, the mesh representation was colored with velocity. Since this is not a requirement for the vortical structure, we removed this velocity component to achieve the smallest size possible for the extracted representation. Figure 2.3 clearly shows that Q-criterion consistently results in smaller mesh sizes than vorticity magnitude regardless of the cube size. This is likely due to the fact that since Q-criterion does not create a mesh where shearing occurs, thus there are less polygons included in the final representation.

Figure 2.3

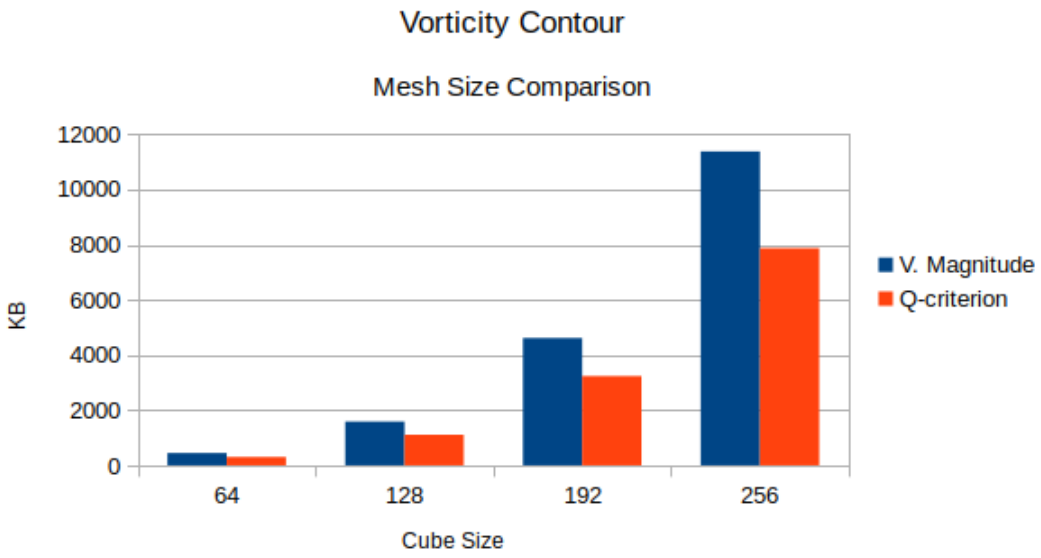

Figure 2.3: Mesh size comparison of Vorticity Magnitude and Q-Criterion

2.2.2 Ghost Cells

When creating iso-surface representations on cubes of data as opposed to the entire dataset, the edges of the iso-surface may be incorrect without additional neighbor cells. These cells outside the boundary of a cube to be processed are called ghost cells. An example of cubes processed without these additional data is shown in Figure 2.4. It is clear that the left side image shows where the two mesh cubes adjoin diagonally from left bottom to right top. There is an overlap on the large vorticity worm on the left, and the smaller right vorticity worm appears to mismatch.

For a 256^3 dataset, the side of the adjoining neighboring cube contains data required to correctly compute the mesh. Therefore, a cube of velocity required to compute a 256^3 mesh would require a 258^3 velocity dataset.

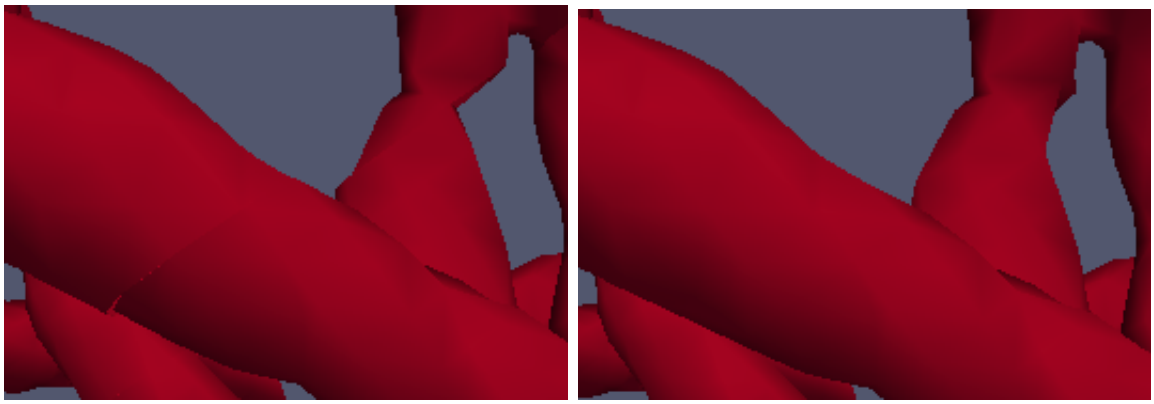

Figure 2.4: Two adjoining mesh cubes, one with no ghost cells (left), one with 1 ghost cell (right)

2.2.3 Q-Criterion

Based on the metrics of speed and extraction size, Q-criterion is the best fit for vortex structure extraction for isotropic turbulence simulations. In practice it produced smaller files and was computed slightly faster than vorticity magnitude. Therefore, for the remainder of this dissertation, Q-criterion will be used to calculate vortex structures.

2.3 Velocity Extraction

Creating mesh representations of vorticies via Q-criterion provides visually appealing extracted data that is reduced in size, however there are issues with these data. First, mesh data required ghost cell processing. Regardless of how this is handled, file size or communication overhead will occur when parallel processing chunks of data. Each cube will require additional data either by having it written out by

the simulation, or by sharing it among processes during the extraction. Second, the mesh presents good visualization, but does not capture data contained *within* these high vorticity regions. In order to solve both of these problems, we present a method of extracting velocity data in and around regions of relatively high vorticity. Since we extract only velocity data there is no iso-surface generation and thus ghost cells are not required.

In order to capture the velocity data, we create a three-dimensional stencil that encompasses the regions of high-vorticity. This stencil masks out low Q regions and generates a sparse representation of velocity data within the regions. This sparse representation is a vtkUnstructuredGrid that consists of floating point coordinates in real space and the corresponding velocity vector at each point, thus each point contains six corresponding floating point values. Since the representation of vorticies in isotropic turbulence appear as worms, the goal is to capture velocities within all points in these worms, while discarding the velocity data outside of these structures. This data is then losslessly compressed to preserve the original values.

We begin by creating a stencil that "cuts out" high-vorticity regions from the full data i.e. points above the Q-criterion threshold. These regions are then dilated to include nearby points that are below the threshold. Dilating by four cells allows us to later compute most quantities of interest, including Q-criterion, vorticity magnitude, marching cubes for iso-surface extraction, velocity derivatives, and 4th-order Lagrangian interpolation. In order to create the stencil we create a bitmask dataset

of the same dimensions of the original dataset and set all values above the threshold to one and those below to zero. Next we dilate this stencil with kernel size of four, meaning that each point that is already set to one sets all points within four voxels to one. Then we mask the velocity field with this zero/one data set, which extracts velocity values from the high vorticity regions and zeros out all other regions. The resultant data set contains a subset of velocity where each velocity vector retained contains a point coordinate to define its spatial location. The data can be utilized to reconstruct Q-criterion and iso-surfaces at or above the specified threshold. Figure 2.5 illustrates a visualization of dilated velocity volume utilizing the Q threshold of 783. Figure 2.6 demonstrates the ability to extract contours at higher thresholds from the thresholded velocity volume shown in Figure 2.5.

2.3.1 Extraction process

In order to perform the extraction, we utilize the Visualization Tool Kit (VTK).[32] First, the raw data is read from the burst buffer or disk and put into an array. This array is converted to a vtkFloat array. Next we create a vtkImageData object and set the extent and spacing to match the dimensions of the cube being processed. Then we add the vtkFloat array to the image as a vector array. Once the vtkImage is created the filter is then chosen depending on which vorticity calculation is requested. For vorticity magnitude, the vtkCellDerivatives filter is used and for Q-criterion, the vtkGradientFilter is used. The vtkCellDerivatives filter creates a vector array of vor-

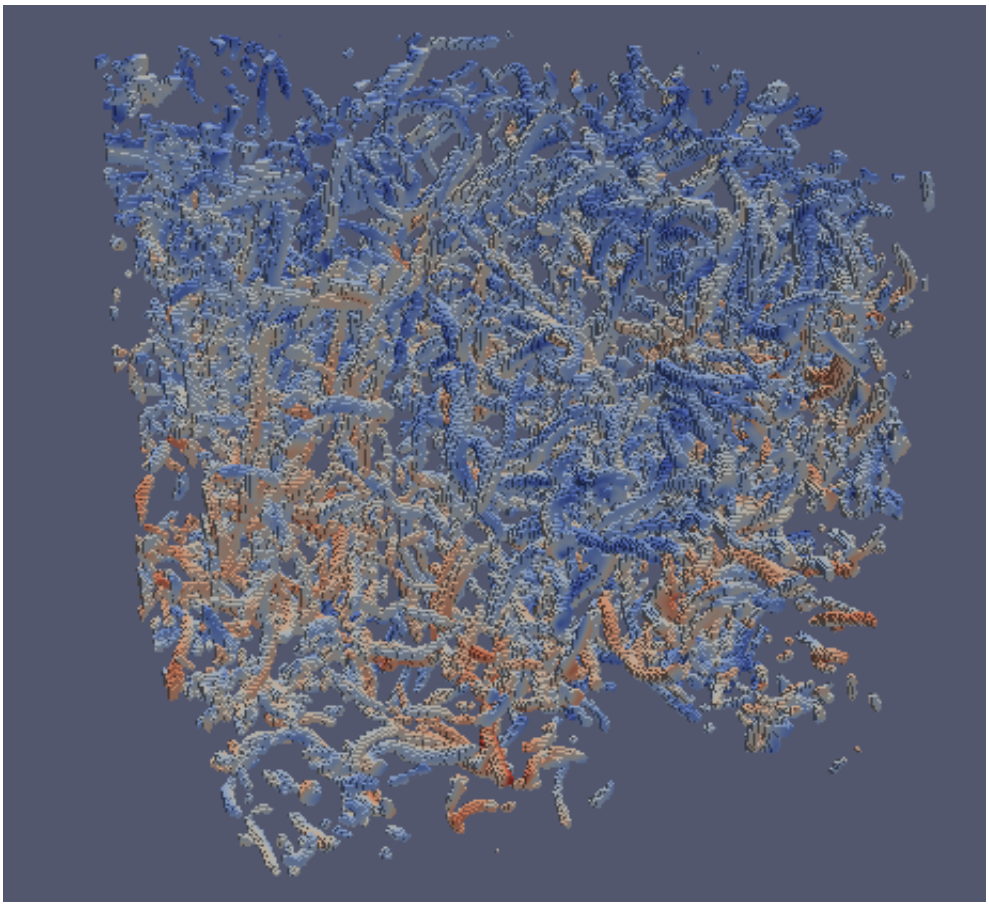

Figure 2.5: Visualization of a 256 cube of dilated velocity in regions above Q threshold of 783

ticity, therefore the vtkImageMagnitude filter is utilized to generate the appropriate vorticity magnitude values.

Once the Q-criterion or vorticity magnitude is computed, the vtkCountourFilter can be utilized to generate a visualization of the vorticity at a specified threshold. The output is a set of polygons in the vtkPolyData format. VTK also provides the ability to render the data to view immediately in 3D, or render off screen with a camera angle to take a snapshot of the data.

Once the optional visualization is complete, the next step is to create a vtkImageThreshold object with the input being the output from the Q-criterion filter or the vtkImageMagnitude filter. Next we set the threshold on the vtkImageThreshold and set all points above the threshold to one, and all other points to zero. This creates an image object of the same dimensions of the original data that is a bitmask stencil where vorticity meets the threshold. Then we run the vtkImageDilateErode3D filter on the bitmask with a kernel size of four in x, y, and z direction. Then we direct the filter do dilate all one values by the kernel size. The output from this operation is a dilated bitmask we will use as a stencil on the velocity data. We take the bitmask and create a stencil using the vtkImageToImageStencil filter. The stencil is then applied to the original velocity data to provide the final output. The result is a structured grid of velocity values at the dilated threshold points, and all other values set to zero. Optionally we can create an unstructured grid by removing all the zero values, which may reduce the size depending on the threshold that is set.

2.3.2 Extraction Speed and Size

We perform a threshold and dilation velocity cutout operation on a cluster with SSD burst buffers that contains a single timestep of raw simulation data. We vary the cube size into which we decompose the problem in order to find the cube size that maximizes throughput. Smaller cubes reduce I/O throughput and reduce skew and memory pressure. Larger cubes increase I/O throughput, but reduce the effi-

cacy of caching, particularly on smaller processor caches up the memory hierarchy. We find that a cube size of 256^3 maximizes throughput for this computation (Table 2.2). Above 192^3, performance is stable and degrades slightly above 256^3, which we attribute to increased cache misses.

Size	Read	Q	Thresh	Write	Total	Throughput
64	.029	.117	.0154	.0266	.222	13.51 MB/s
128	.043	.877	.064	.136	1.34	17.91 MB/s
192	.080	2.83	.206	.542	4.46	19.32 MB/s
256	.136	6.15	.399	1.08	9.522	20.17 MB/s
384	.373	20.68	2.23	5.67	34.337	18.87 MB/s
512	.788	48.76	5.31	13.17	78.86	19.47 MB/s

Table 2.2: Comparison of I/O and computation times in seconds when processing a single cube

Averaged over all cubes, the extracted thresholded velocity data is reduced by a factor of 29 times. The raw size of 256^3 of velocity data is 192 MB and the dilated extraction averages 6.7 MB.

The reduction factor is directly related to the Q Criterion threshold selection. In Figure 2.7, we present a graph that is logarithmic in the size axis to show the non-linear relationship between the two.

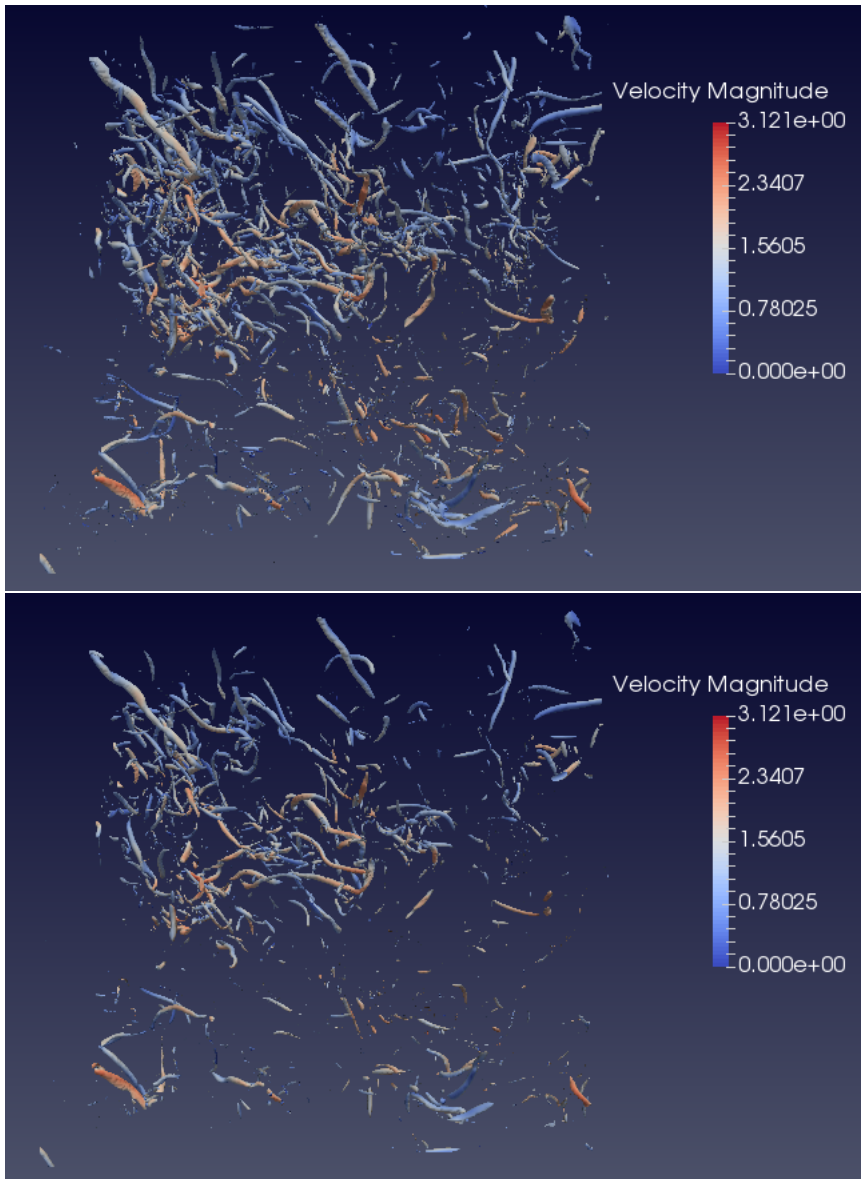

Figure 2.6: Iso-surface extraction from dilated velocity threshold at Q thresholds 1700 (left) and 2500 (right)

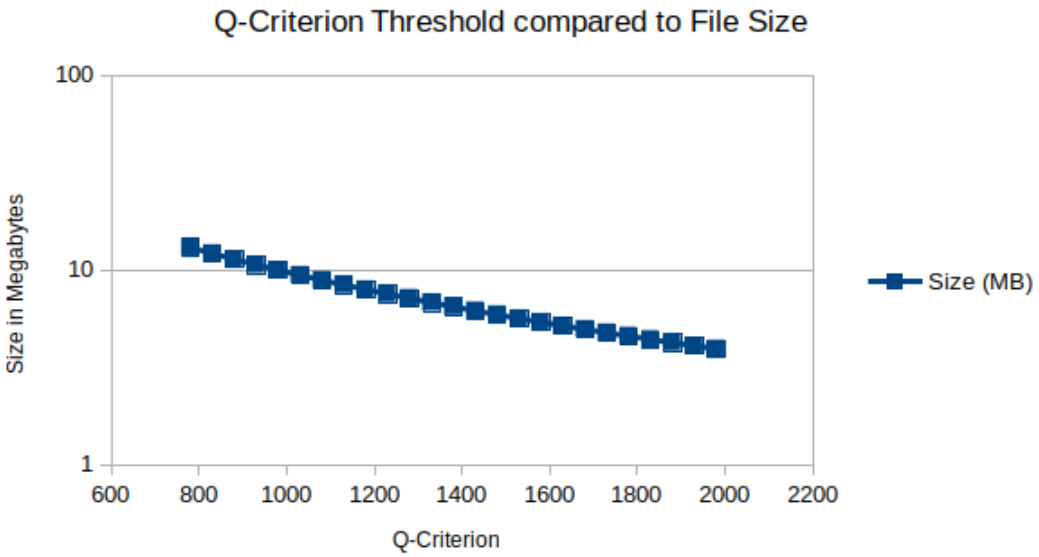

Figure 2.7: Relationship between cube file size and Q Criterion threshold

Chapter 3

Compression

In Chapter 2, regions of relatively low vorticity were deliberately removed in order to reduce the size of the final representation. While thresholding works well for scientists studying events specifically within extreme vortical regions, it may be necessary to save information outside those regions for post analysis. For example, vortex precursors may occur in initially weak vortical regions, which then act as seeds for subsequent vortex intensification. In addition, a researcher may need information about conditions where velocity may be relatively high, which may not be contained in our thresholded data due to the fact that vorticity is a measurement of curl or rotation. We present a method for storing *all* of the data in a lossy compressed form for post analysis and visualization. This does not provide exact raw simulation data, however, it provides data that is within a defined error tolerance. The error introduced on these data will be shown to be insignificant for the purposes of visualization,

making the data desirable for post visualization analysis. A lossy compression algorithm can potentially provide an order of magnitude or more reduction, whereas a lossless compressor cannot achieve these results.

3.1 zfp

In order to store the data in a lossy form, we utilize a recent compression algorithm, zfp,[26] designed specifically for the compression of multi-dimensional, floating-point scientific data. It is an open source C/C++ library for compressing floating-point arrays that support very high throughput read and write random access. It contains various options for compression, one of which is to specify an absolute error tolerance to provide error bounded data. Utilizing this method at an error tolerance of 10^{-1} on the velocity data (the root-mean-square value of the velocity fluctuations is 0.686 while its mean is zero), we achieve an effective reduction of one order of magnitude from the raw velocity data. The tolerance is the maximum difference between the original value and the decompressed value after zfp compression. We note that this reduced dataset is intended for post analysis and visualization, and cannot be used as checkpoint data to restart the simulation. zfp also provides methods of in-memory compression, however this functionality is not required for the extraction application.

Specifically, zfp provides a fixed-rate compression scheme by using blocks of 4^d values where d is the dimensionality of the dataset. Since turbulence data is in 3

dimensions, it is compressed by zfp in 4^3 blocks. It performs five steps to achieve the final compressed representation:[26]

- Align the values in a block to a common exponent
- Convert the floating-point values to a fixed point representation
- Apply an orthogonal block transform to decorrelate the values
- Order the transform coefficients by expected magnitude
- Encode the resulting coefficients one "bit plane" at a time

During the encoding operation, an embedded coding scheme is utilized to encode the transform coefficients. This is where the loss tolerance is applied in the algorithm. The embedded encoding can truncate the bit stream at a defined level which simultaneously degrades fidelity and reduces representation size.

3.1.1 zfp utilization in VTK

We extended the Visualization Toolkit (VTK)[32] by creating a wrapper for the zfp compression library, and added the zfp compression library to VTK. Since zfp was designed to work on dense datasets, we chose to add the algorithm to the vtkImageData object. The vtkImageData object stores a structured grid with up to three dimensions. We also needed to make zfp work on data that contains more than one component since turbulence velocity data is in three components, namely ux, uy, and

uz. The library provides a striding option, which we utilized to separate ux, uy, an uz. Then each component is compressed separately and each compressed component is concatenated in its binary form. The data sizes for each axis are stored within the VTK XML file format as metadata in order for VTK to correctly separate the components and decompress the data. During decompression, each component is decompressed into a separate array and interleaved back to their original representation creating a VTK float array of velocity vector values. If pressure or another scalar field were added, this would be compressed without the need for the interleaving process.

3.2 zfp Analysis

To begin performance analysis of zfp, we compressed isotropic turbulence velocity data with a predefined error-bounded tolerance of 10^{-1}. This means the absolute maximum difference between the original value and the value after zfp decompression can be no greater than .1. We found that this absolute maximum was never reached and the error was more than 6 times smaller than .1 on values that demonstrated the most error. At this tolerance we achieved an order of magnitude of compression with visually lossless reconstruction, which is far superior to the default ZLib library utilized in VTK.

Table 3.1 shows the resulting size of compressing different sized cubes of isotropic turbulence data, along with the amount of time required to compress the cube.

Cube Size	Raw Size	zfp Size	Total time (s)	Reduction	Throughput
128	25 MB	2.3 MB	.334	x10.9	74.85 MB/s
192	81 MB	8.1 MB	1.05	x10	77.14 MB/s
256	192 MB	18 MB	2.09	x10.7	91.87 MB/s

Table 3.1: zfp Compression by cube and time

3.2.1 Compression Speed and Reduction

The VTK 7.1 release contains ZLib and LZ4 lossless compressions schemes, therefore we compared these two compression algorithms alongside zfp. In order to provide performance metrics, we tested speed and ratio of original data size to compressed representation size. The speed results are located in Figure 3.1, and the compression results in Figure 3.2. ZLib was significantly slower than the other algorithms but provided the best *lossless* compression. The zfp compressor proved to be just slightly slower than LZ4, but provided superior compression over both lossless compression algorithms. LZ4 proved to be the least useful with our data, since it did not provide any significant compression.

These results clearly show zfp is fast and compresses isotropic turbulence velocity data exceptionally well. However, since it is lossy we must analyze how much error is introduced during compression. We compared sample of velocity from the 64^3 original data and the same cube compressed and decompressed by zfp in order to introduce compression loss. The result was a root mean square error of .00262, .00263, and .00263 for ux, uy, and uz respectively for 262,144 points. In this same dataset, we found the minimum difference to be 0 for a handful of points, which is essentially

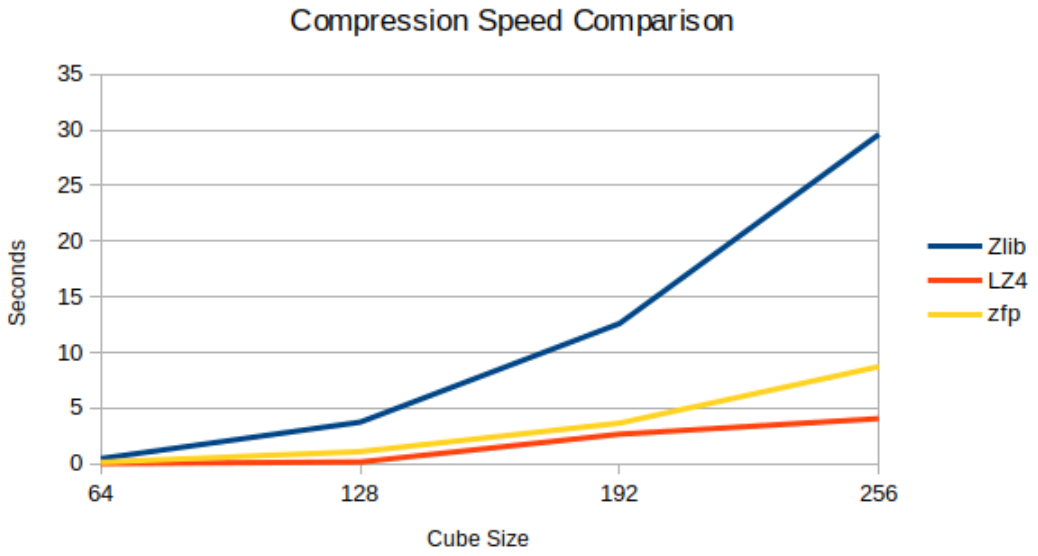

Figure 3.1: Speed comparison of Zlib LZ4 and zfp compression on multiple cube sizes

lossless. We found the maximum difference from the original velocity and the zfp compressed velocity to be .015, which is not only within the .1 bounded tolerance, but nearly a magnitude less. Since zfp uses truncation on the bitstream to guarantee error less than .1, the actual error depends on the data that is being compressed. In this case, isotropic turbulence velocity data has gradual variance between points which results in lower error between the original data and compressed data.

3.2.2 Visual analysis

To perform the analysis, we saved each cube as a VTK Image Data file which uses a few lines of XML for metadata about the object (for example, dimensions and array names) and a VTK float array that is compressed using zfp and saved as

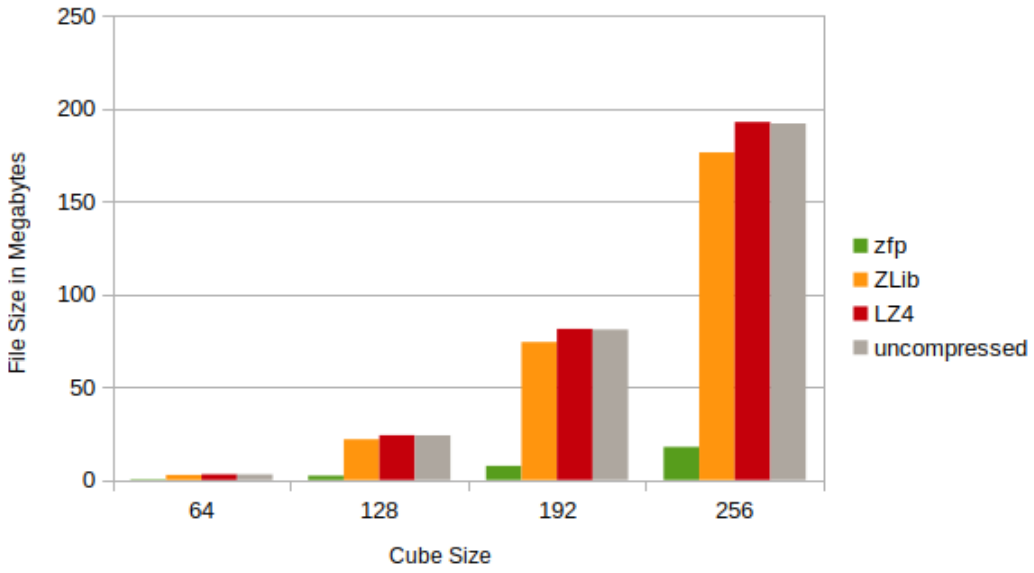

Figure 3.2: Size reduction comparison of Zlib LZ4 and zfp compression on multiple cube sizes

binary appended data to the XML file. This metadata is extremely small and has no significant impact on overall file size. Figure 3.3 shows a surface representation of a 256 cube of velocity data. The left figure is the raw velocity magnitude, while the right figure was compressed by zfp and then decompressed for visual representation. The two cubes are indistinguishable in this figure and also when viewing at all zoom levels.

3.2.3 Visual analysis on derived fields

Since it is clear that raw compressed zfp data at .1 tolerance was visually indistinguishable, we performed a vorticity computation using Q-Criterion from zfp decompressed data and subsequently contoured the data. While the data in Figure

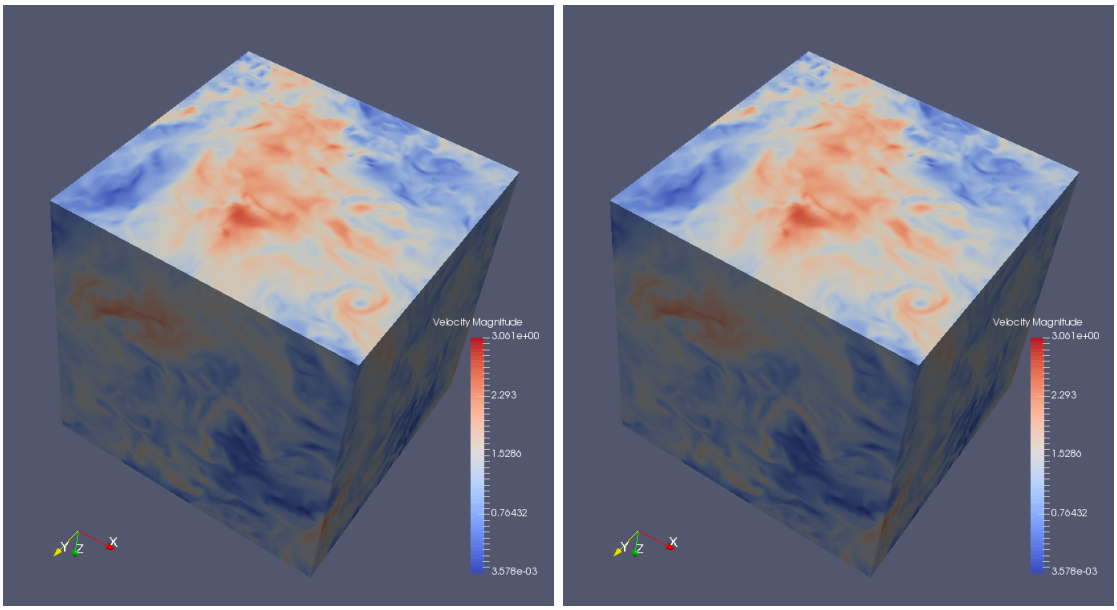

Figure 3.3: Isosurface of a 256 cube of isotropic turbulence velocity data. Left: Raw velocity. Right: zfp Compressed at 10^{-1} tolerance.

3.3 was indistinguishable at various zoom levels, the contour does show very minor artifacts introduced by the error in zfp. Figure 3.4 contains two contours with velocity coloring. The top image was created from original velocity data and contoured at a threshold of 783 Q-criterion. The bottom image was velocity data compressed by zfp, then decompressed and contoured at 783 Q-criterion. Initially they appeared indistinguishable, but once the zoom level was changed minor artifacts became present. For example, the large vortex in the upper right corner has minor crease in the top image, and it is more pronounced in the bottom image. The results of this visualization demonstrate that zfp is acceptable for storage of velocity data when utilized for visualization of Q-criterion.

3.2.4 Velocity Extraction Compression

In Chapter 2, we performed a dilated velocity extraction in high vorticity regions using Q-Criterion. By default, we saved this data using the standard ZLib compression library. In this section, we perform a compression size and visual comparison of a 3D volume rendering of the dilated velocity using original extraction data versus data that was compressed using zfp. Since zfp only works on a structured grid, the representation used was a structured grid with zero velocity in areas of low vorticity. Figure 3.5 contains the volume rendering of original velocity data and zfp compressed data. Visually they does not appear to be any significant difference between the two renderings. The ZLib representation is 13MB and the zfp representation is 7.6MB thus the effective size is reduced by 41%. This reduction is not as much as the reduction of a dense grid of velocities which was closer to 90%. This is expected since zfp is designed for compression of values that do not have a lot of variance compared to the spatial location. The variance in this extracted data will be significant on the edges of the extracted values since the neighboring points will be set to zero.

Next we performed a Q-criteron contour on the velocity extraction data. Figure 3.6 contains images created by contouring original velocity extraction data and zfp compressed data at a threshold of 1174. Visually these two images look the same. We did notice that if they are superimposed on each other, there are very slight minor variations that become visable. This is expected since there is loss with zfp, but for the application of visualization it does not appear to be an issue.

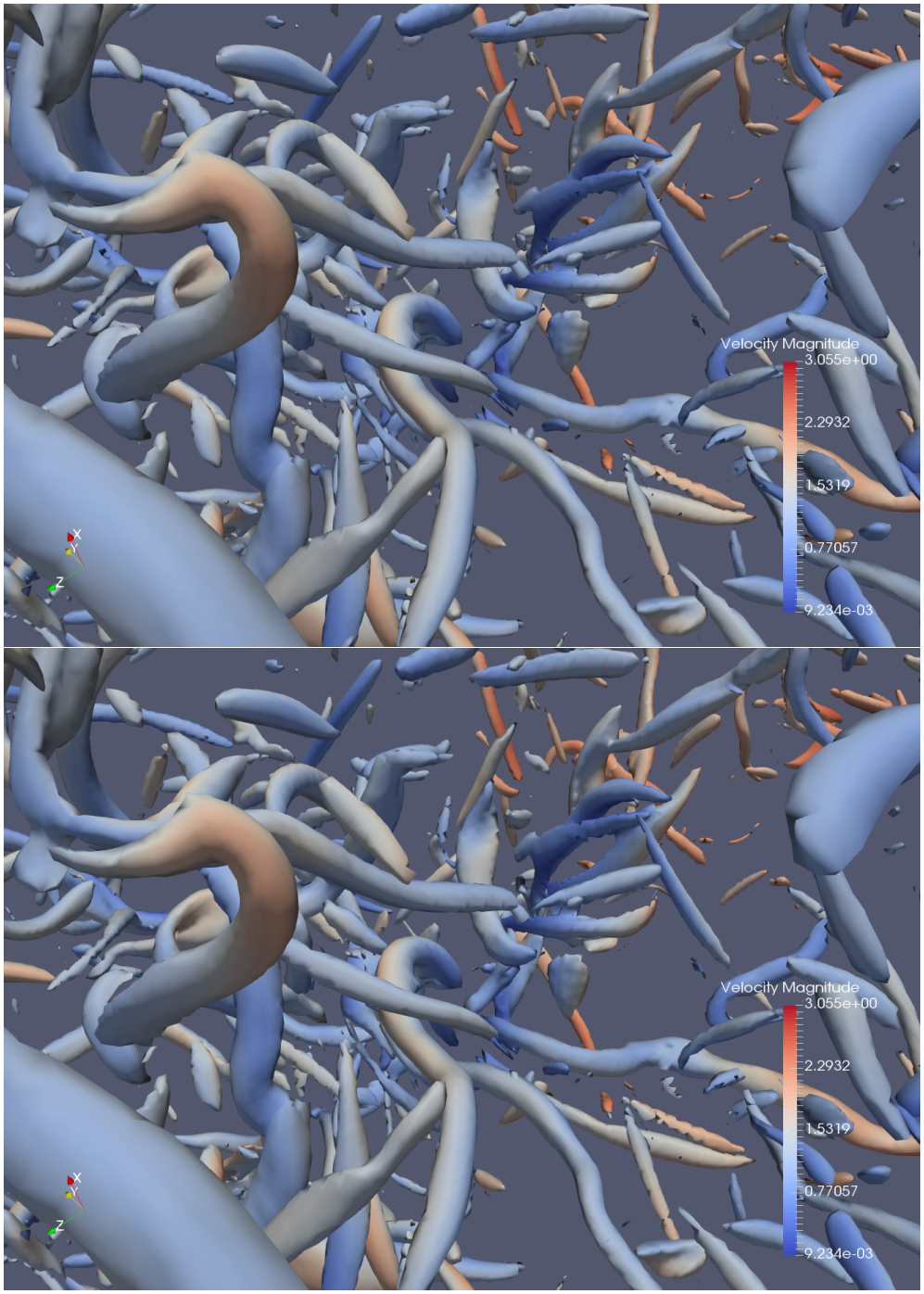

Figure 3.4: Top: Mesh constructed from original velocity data. Bottom: Mesh constructed from zfp compressed data with .1 tolerance.

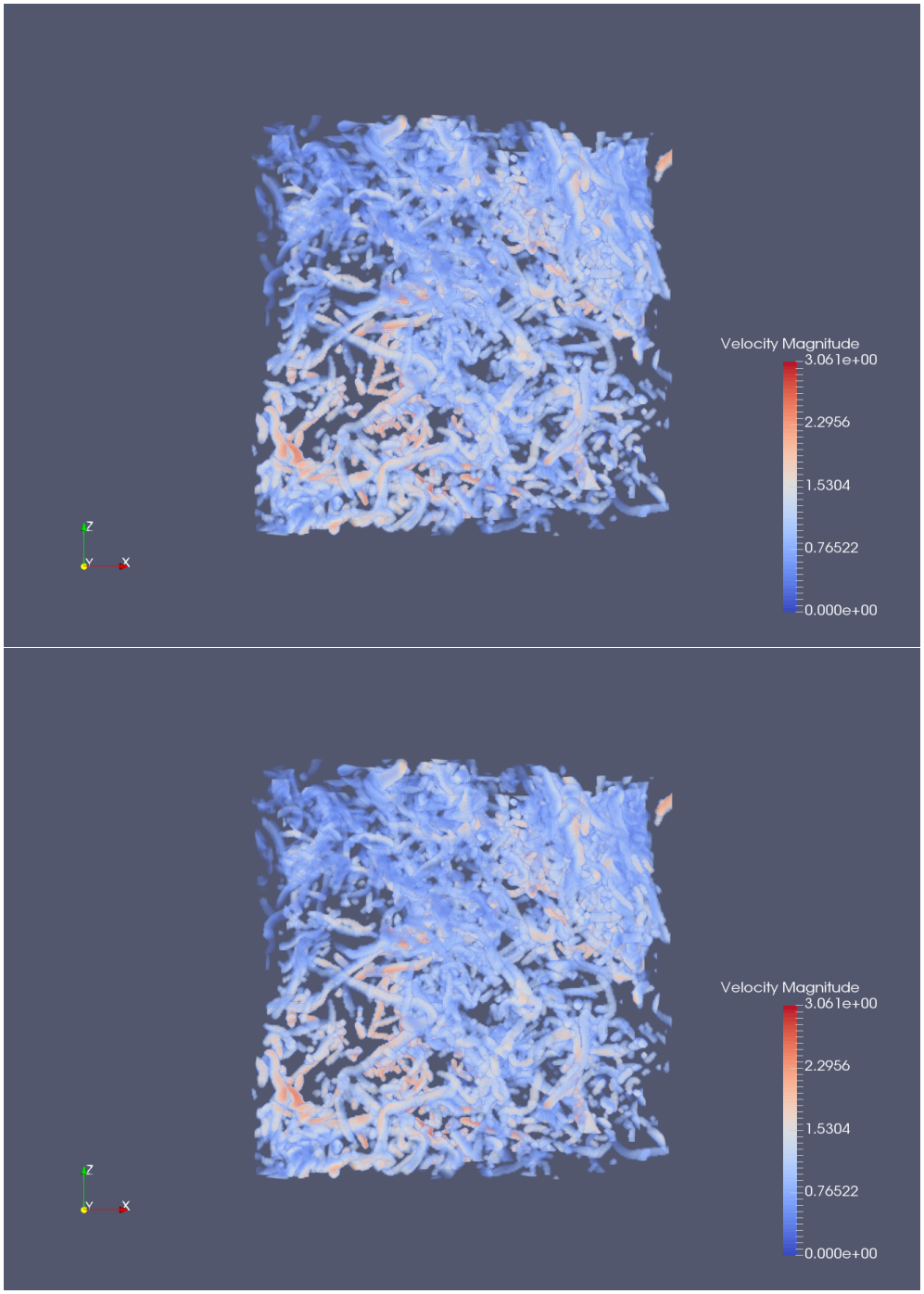

Figure 3.5: Top: Volume rendering of original velocity extraction. Bottom: Volume rendering of zfp compressed velocity extraction.

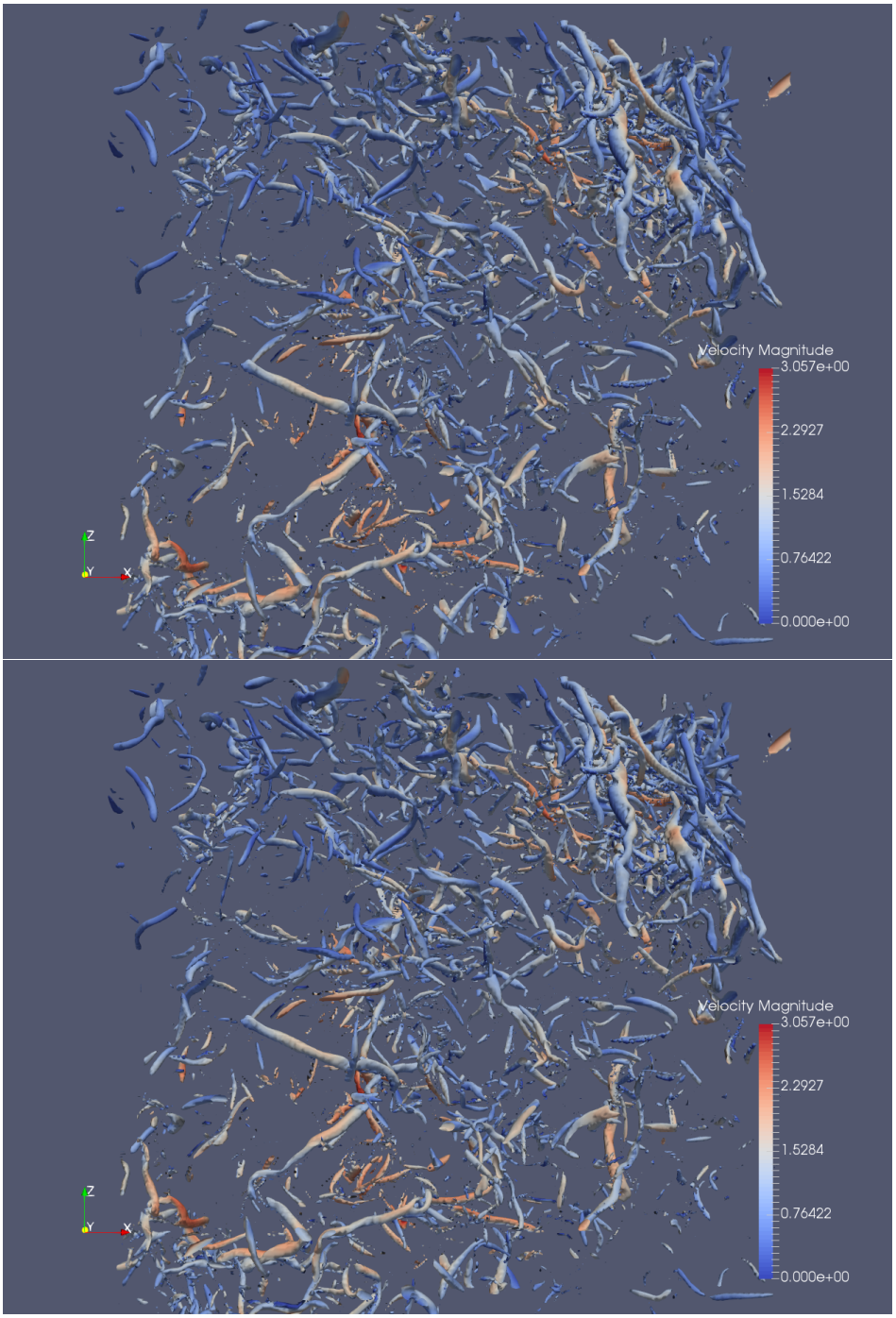

Figure 3.6: Top: Contour from original velocity extraction. Bottom: Contour from zfp compressed extracted velocity.

Chapter 4

Myrcene

Myrcene is a system we created that contains a coordinator and client that performs distributed parallel data extractions on a cluster of nodes. The name myrcene was chosen because it is an natural organic hydrocarbon that is an essential oil in several plants to include bay, hops, and thyme. Myrcene is extracted from plants for its pleasant odor and flavor for various uses. In brewing beer, myrcene is extracted from hops during a turbulent boil. Therefore, we called our system Myrcene since it helps scientists perform extraction of essential data, and in our case turbulence data. Once a scientist determines which data must be extracted from a direct numerical simulation, they need to design a method of running the extraction algorithm in a distributed and parallel environment. This can be done with various scripts, but requires timing and synchronization to run correctly across several nodes. Myrcene simplifies this process by allowing a scientist to write a module that takes in parameters of the

data size to perform extraction on a single piece of data. This module is added to the client, then the client can run on any number of nodes available for extraction. The coordinator in Myrcene contains a database of nodes and will communicate with the client to synchronize extraction of data in parallel on each node. The coordinator also provides synchronization between the simulation and the clients. In addition to simplifying the extraction process, it also collects statistics about the extraction and automatically generates extraction time charts.

4.1 Related Work

This work automates and simplifies parallel and distributed programming and operation. While we specifically address data extraction from the burst buffer architecture, others have devised methods of simplifying parallel distributed programming. Blanas et al. present a system called Scientific Data Services (SDS/Q)[15] which provides a query interface for the Hierarchical Data Format version 5 (HDF5) that runs in a parallel distributed environment. This work is very similar in that it can operate on any type of scientific data stored in the HDF5 format and abstracts the parallel and distributed operation. However, the SDS/Q system differs from Myrcene in that it was designed for querying as opposed to computationally-intensive feature extraction and visualization. The goal of the SDS/Q system was to create a system that would outperform relational database systems. Tournavitis and Franke[16] present a

semi-automatic method of compiling applications written as single thread into a multicore parallel application on a single node. This is a very generic approach since it can work on any code, however it does not address the distributed architecture of a cluster.

The automated parallelization of functions in Myrcene was inspired in part by Map/Reduce,[17] which has been reimplemented and extended by many parallel systems such as Hadoop[18] and Spark.[19] Myrcene inherits the notions of data-parallel execution and functional parallelism. Unlike Map/Reduce which performs sequential I/O, Myrcene uses the burst-buffer SSDs to support arbitrary data access patterns. Fast I/O for SSDs have been used as a building block for graph-processing[20] and linear algebra[21] systems. These are again limited to parallelism within a single node. Pearce et al.[22] demonstrates a distributed implementation of graph-analysis on SSDs. This is specific implementation of graph traversal algorithms and not a general execution framework.

4.2 Design

Myrcene contains a feature extraction coordinator (FEC) and a modular expandable command line client that performs parallel operations on burst buffer and/or extraction nodes. The FEC contains a database backend that stores node information and all metadata required for execution. The FEC serves as the central automatic

coordinator with all nodes utilized in each extraction. A diagram that depicts the coordination between the clients, FEC, and simulation nodes is in Figure 4.1. The

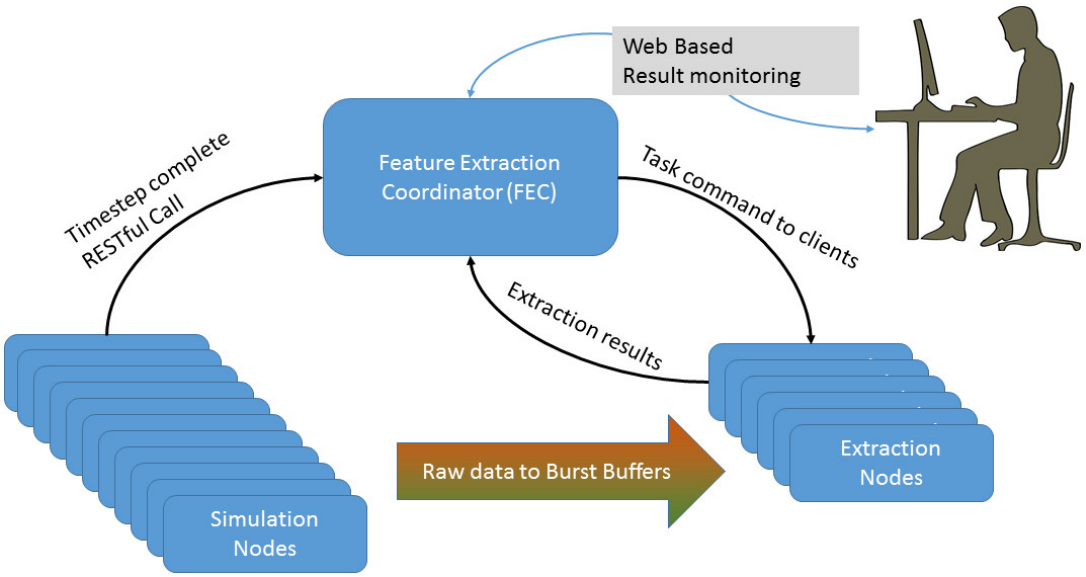

Figure 4.1: Myrcene

FEC database stores node specific information which includes number of cores to run the extraction in parallel and the hostname of each node. The FEC database also contains jobs which are the execution point for a simulation. When the simulation finishes writing a timestep to the burst buffers, it must make an HTTP request to the FEC with the timestep number that completed. A job contains one or more tasks that contains filename and metadata information specific to the dataset, the module type, cube sizes, and multipurpose generic fields that can be used for module specific required parameters. An entity relationship diagram of the FEC database is depicted in figure 4.2

One key benefit inherent to the client is that it loads all the libraries (VTK,

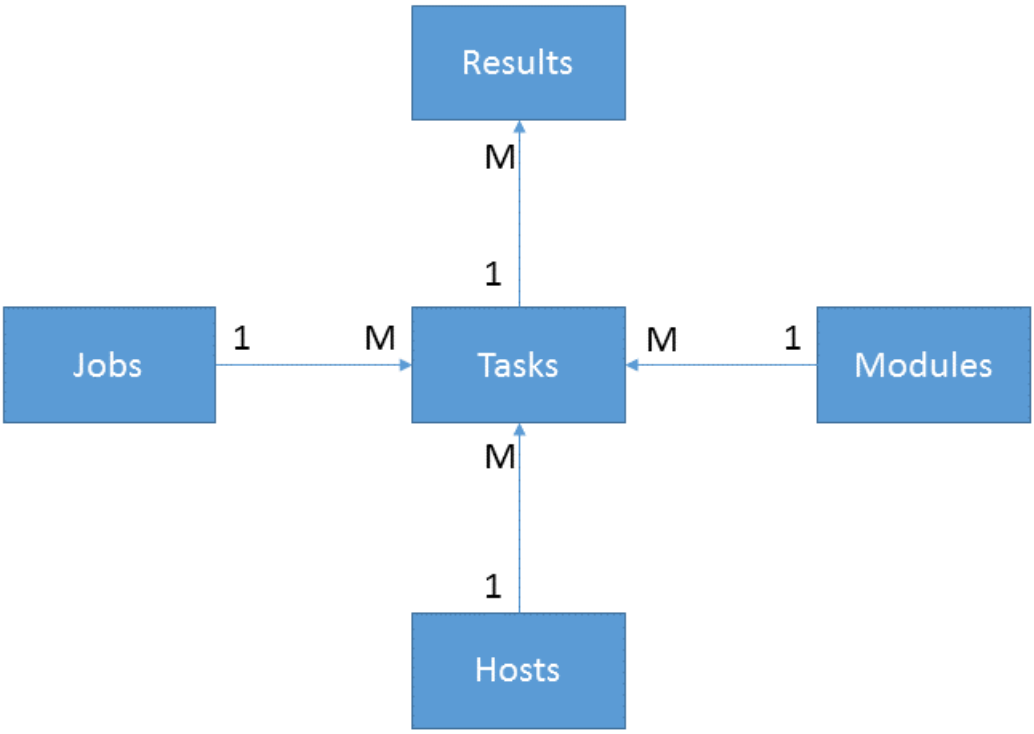

Figure 4.2: Myrcene

HDF5 and any additional module libraries required) at runtime and maintains them in memory throughout the entire simulation. This gives an advantage over a scripted implementation that loads the libraries each time data is ready for extraction. The amount of time required depends on what modules are required, but as an example of how much time it may take, the VTK library takes approximately 5-15 seconds to load into memory depending upon the node configuration.

4.3 Distributed Parallel Operation

The FEC serves as the key synchronization point for the entire extraction process. Once the simulation completes writing a timestep of simulation data to the burst buffer nodes, a single Representational State Transfer (REST)[33] call containing the timestep and job number is made to the FEC. This single RESTful HTTP request initiates the extraction process. Once this call occurs, the FEC distributes a JavaScript Object Notation (JSON) encoded packet to every node that has a task assigned to it from the job. This packet includes data shape information, module selection, additional module specific parameters, and number of processes to spawn for the extraction. The clients receive the packet and begin a parallel extraction of data based on the number of processes specified in the FEC database for each specific node. Upon completion of the task, the clients return a JSON encoded packet with a success or failure along with timing metrics gathered from the extraction process. If the module generates imagery for visualization, the images are stored on shared storage that the FEC can access for web-based viewing. Once the FEC receives a client packet, it creates a result record in the database for the node and stores all metadata about the extraction process. The dynamic report generator can then read from the results table to generate a graph depicting the results of the operation. Therefore the data scientist can view the performance of the extraction immediately as nodes report completion of their tasks. This occurs once the first timestep is extracted, and the data will continue to build up the graph as the simulation continues.

4.4 Modules

All modules receive the same basic information from the FEC. At a minimum this includes the input and output filenames and locations and the dimensions of the data cubes to be processed. Additional parameter fields are provided for specific information unique to each module. For example, a threshold value can be passed to a module that performs thresholding, whereas a lossy compression module may need a tolerance value that is unique to the module. The modules and client should be loaded and executed from shared storage to enable code updates without the need to redistribute code to each individual node. In practice we utilized the srun command from the Slurm[34] workload manager to initiate the client simultaneously on all burst buffer nodes.

4.4.1 zfp

The zfp module is a generic dense grid (1D, 2D, or 3D) lossy compressor for scientific floating point data. The zfp compression algorithm was specifically designed to compress floating point data where the values spatially near each other on the grid have low variance.[26] One key feature of zfp is the ability to provided bounded error rate (tolerance) for the compression. This allows a domain scientist to specify how much tolerance for error they are willing to accept in order to achieve the maximal compression with a guarantee of how much values shift from the original data. In

order to standardize the file format, we chose to expand the Visualization Toolkit (VTK)[32] to include the zfp compressor by providing a zfp wrapper for VTK. This allows a user to save the zfp compressed representation as a standardized VTK Image Data file (.vti). Our wrapper code is currently being merged into VTK 7.1 which will make zfp available to all VTK users in the future. While we have built this module to work with VTK, a zfp module could be written to save the raw data as binary zfp data if VTK functionality is not required. This module specifically relies on dimension size of the data for compression, and this is provided to the module from the FEC rather than determining the size during runtime. This allows the data format to be completely raw simulation data which would not contain any metadata.

4.4.2 Vorticity Mesh

This module creates a sparse representation of vorticies by using a polygonal iso-contour at a chosen threshold. In order to create this representation, the module reads in raw velocity data from the specified location (ex. burst buffer) and computes a vorticity or Q-criterion magnitude. Using the magnitude, the marching cube algorithm[35] is applied to perform a contour where the magnitude meets the specified threshold. This mesh representation of vorticies provides excellent visualization of the dataset. In isotropic turbulence vorticity appear as worms when the contour algorithm is applied. In addition to creating excellent visualization, it also reduces the dataset significantly since the mesh is composed of 2D polygons that reconstruct the

surface of vorticity as opposed to saving the entire dense dataset of velocity points. A sample of data reduction is provided in Table 4.1.

Cube Size	Threshold	Raw Data Size	Mesh Size	Reduction
64	59.975	3MB	1011KB	3x
128	59.975	24MB	2.56MB	9.4x
256	59.975	192MB	26.86MB	7.15x

Table 4.1: Data reduction by cube size and threshold value

Figure 4.1 shows that the data required to contour the high vorticity regions with colored velocity results in significant data reduction. Due to the marching cube algorithm properties,[36] ghost cells outside of each cube are required in order to create a proper iso-contour. In order to assist with this problem, the number of ghost cells required is passed to the module. This allows the module to trim out the ghost cells before the image data is saved. The only requirement in this case is for the simulation output to be configured to provide the additional ghost cells for each cube in order for the contour to be correction computed on the edges of each cube.

4.4.3 Vorticity Dilated Volume extraction

Creating a polygonal mesh representation of vorticies provides an excellent visual representation of vorticity, however it does not capture the raw velocity data *within* the high vorticity regions (data inside the worms). While identifying where in 3D space high vorticity regions exist could lead to further scientific discovery, not having velocities within these regions prevents the scientist from performing further analysis

or performing computation on the original velocity data. In fact, P. K. Yeung, X. M. Zhai, and K Sreenivasan[27] describe "extreme events" that occur inside and around these high vortical regions where energy dissipation and squared vorticity (enstrophy) are orders of magnitude higher than the mean values. Therefore it is essential to collect the velocity data in and around these regions in order to capture these extreme events. This module solves this problem by extracting velocity in and around the high-vorticity regions. The extraction result is a three-dimensional irregular grid (or optionally a dense grid with zeroed out values in low vorticity regions) that encompasses the high-vorticity regions. This grid masks out low vorticity regions and generates a sparse representation of velocity data within the high vorticity regions. In isotropic turbulence vorticity appear as worms, therefore the module will capture velocities within all points within these worms, while discarding the velocity data outside of these structures.

The extraction module begins by creating another data set in which points above the Q-criterion (or optionally vorticity magnitude) threshold are set to one, using the specified threshold, and all other points to zero, thus creating a 3D stencil. Next the module dilates the stencil by setting zero values neighboring one values to one with a specified kernel size of (default 3), such that any zero value that is within three voxels from a one value is set to one. Then the velocity field is masked with this zero/one stencil, which extracts velocity values from the high vorticity regions. It optionally can convert the resulting non-zero regions into an irregular mesh to remove

the zero values. The resultant data set contains only velocity, but it can be utilized to reconstruct Q-criterion, shear, iso-contours or any other quantity in post analysis, subject to that computations stencil fitting inside the dilated region. In figure 4.3, a 256 cube of velocity volume extracted by the module was utilized to generate a mesh at two different thresholds, which is not possible utilizing an extracted mesh.

This extraction method results in a very significant size reduction of the original data. A 256^3 of velocity data with three components per point is 192MB, and the reduction yields an average size of 6.7 MB. This reduction in size allows the client to write the reduced dataset to shared storage significantly faster than if the entire cube was written to shared storage.

4.4.4 Vorticity Dilated Volume extraction with visualization

This module is a modification of the prior module in the fact that once the vorticity is calculated, a 3D contour is performed and 2D snapshots with different camera angles are rendered and saved as a Portable Network Graphic (PNG) to shared storage. The rendering is performed off screen and will utilize a GPU if present in the node. The module by default provides six perspectives of a cube of data so that all sides can be viewed. Utilizing the module specific parameters, it can provide custom view angles specified by the user. The velocity data in high vorticity regions is also stored like the

previous module. Figure 4.4 contains these six perspectives performed on one 256^4 of velocity data.

4.4.5 Unstructured Grid

This module is unique from prior modules because it directly operates on an unstructured grid. In order to test this module, we utilized a partial dataset from an xRage simulation[37] of an asteroid landing in an ocean. The unstructured grid data was gathered in-situ and saved in a VTK unstrcutred grid format (vtkUnstructredGrid). The representation used in this format is attribute values assigned to points in real space. Therefore each value (i.e. pressure) would be assigned a floating point coordinate in 3D space. The module we built utilizes additional parameters for the component to contour by, and the lower and upper values of that component. Using this data, the module creates a 3D mesh by generating contours and mapping to polygons. Once the data is mapped it is saved as a vtkPolyData file. The resultant data is a mesh that can be used for visualization. The module could easily be extended to provide in-transit visualization analysis by creating PNG files like the previous module. Figure 4.4.5 displays a slice of the mesh from this asteroid data.

4.4.6 Miscellaneous Modules

The modules presented thus far directly performed an extraction from scientific data for use during a simulation. While we have created a test module as a start point for a scientist to begin creating a module, there are other modules we created to aid in various tasks we encountered during our testing. The first is a simple file converter that converts from the HDF5 dataset format to raw python numpy arrays, which can easily be modified to perform any standard file conversion. Another module was a data download module. In order to pre-stage the data for our testing, we utilized this module to download HDF5 data from the JHTDB over the web to the burst buffer cluster.

4.5 Experimental Results

In our first experiment with Myrcene, we prepared 16 heterogenous nodes on the Darwin cluster at Los Alamos National Laboratory. We utilized the web retrieval module to load a set of 256^3 blocks of data on shared storage. Then we used the file conversion module to copy and store the blocks onto each burst buffer node in a raw data files to emulate the file type expected from a turbulence simulation. Each node received 512 256^3 blocks of raw velocity data. Once the data was prepared, we performed a dilated velocity extraction from high vorticity regions four times and the results are in Figure 4.6. Since the nodes are heterogeneous, the final result

times for extraction are varied as expected. Nodes cn45-49 contained CPUs that provided 16 cores, while the other nodes only had 10 cores per node. Once this test was completed, we scaled up the experiment to 32 nodes with burst buffers. This experiment was also performed on the Darwin cluster, however a different subset of nodes were utilized. Nodes in this reservation ranged from 4 cores per node up to 16 cores per node. Prior to execution, we loaded 1024 256^3 blocks of data on each node. The goal for this extraction was to replicate the data size of an 8192^3 grid, which is 32,768 blocks of 256^3. As seen in Figure 4.7, the far right nodes contained only 4 cores which significantly increased their total extraction time.

The dilated velocity vorticity threshold extraction provides original velocity data in these high vorticity regions, however it does not provide any data outside of these regions. In order to capture this data, we used the zfp lossy compressor to gather full field velocity data of the entire dataset. We utilized this compression on the same dataset, and the results are in Figure 4.8. This extraction was performed four times with this configuration. Note that in the first three runs node cn152 had an incorrect configuration for file locations which resulted in a failure. The compression did not occur and no timing result was generated resulting in a zero value on the graph. Once we reviewed the results from the first three runs, we adjusted the configuration to resolve the issue on the fourth run. The issue is easily seen by the graph, which demonstrates that the visual result graph provides instant value to the data scientist. Once errors like this are noticed he or she will be able to quickly narrow down the

error and correct it while the simulation continues to run. We also made adjustments on the number of threads per node on cn190-240, and cn400-405. First we set the nodes to utilize the same number of threads as there were cores per node. In order to determine if we could reduce the computation time for nodes with less cores, we increased cn190-240 from 10 to 16 after the first run (green). Since we noticed a small time reduction, we increased the number of threads from 10 to 16 on nodes cn400-405. The reduced time is due to Intel Hyper-Threading Technology[38] which allows more than one thread to run on each core and execute more efficiently by maximizing computational resources. More specifically if one task is waiting for data, the other task will utilize the computational resources which decreases the overall computation time.

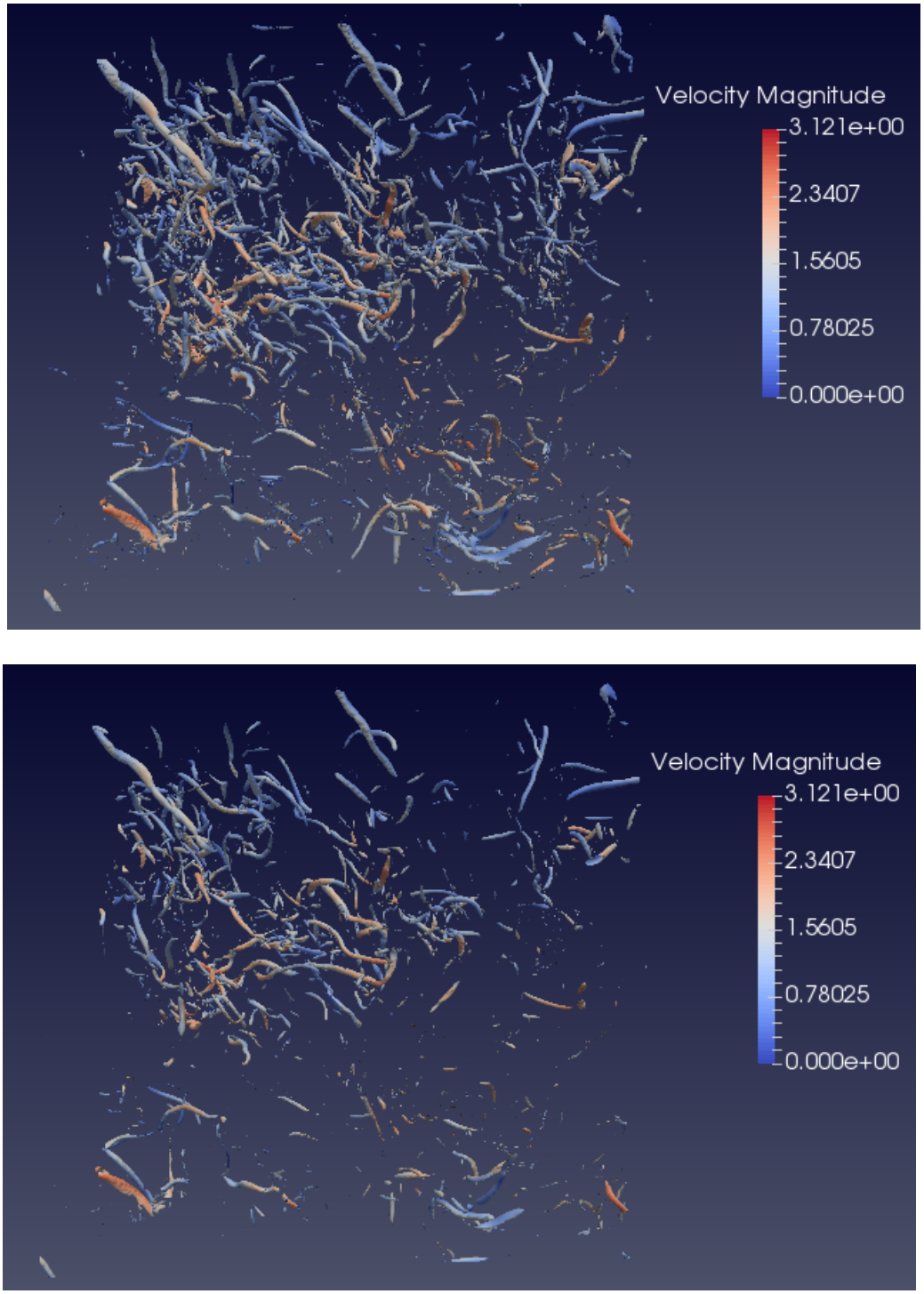

Figure 4.3: Mesh recalculated from a dilated velocity cutout at different thresholds

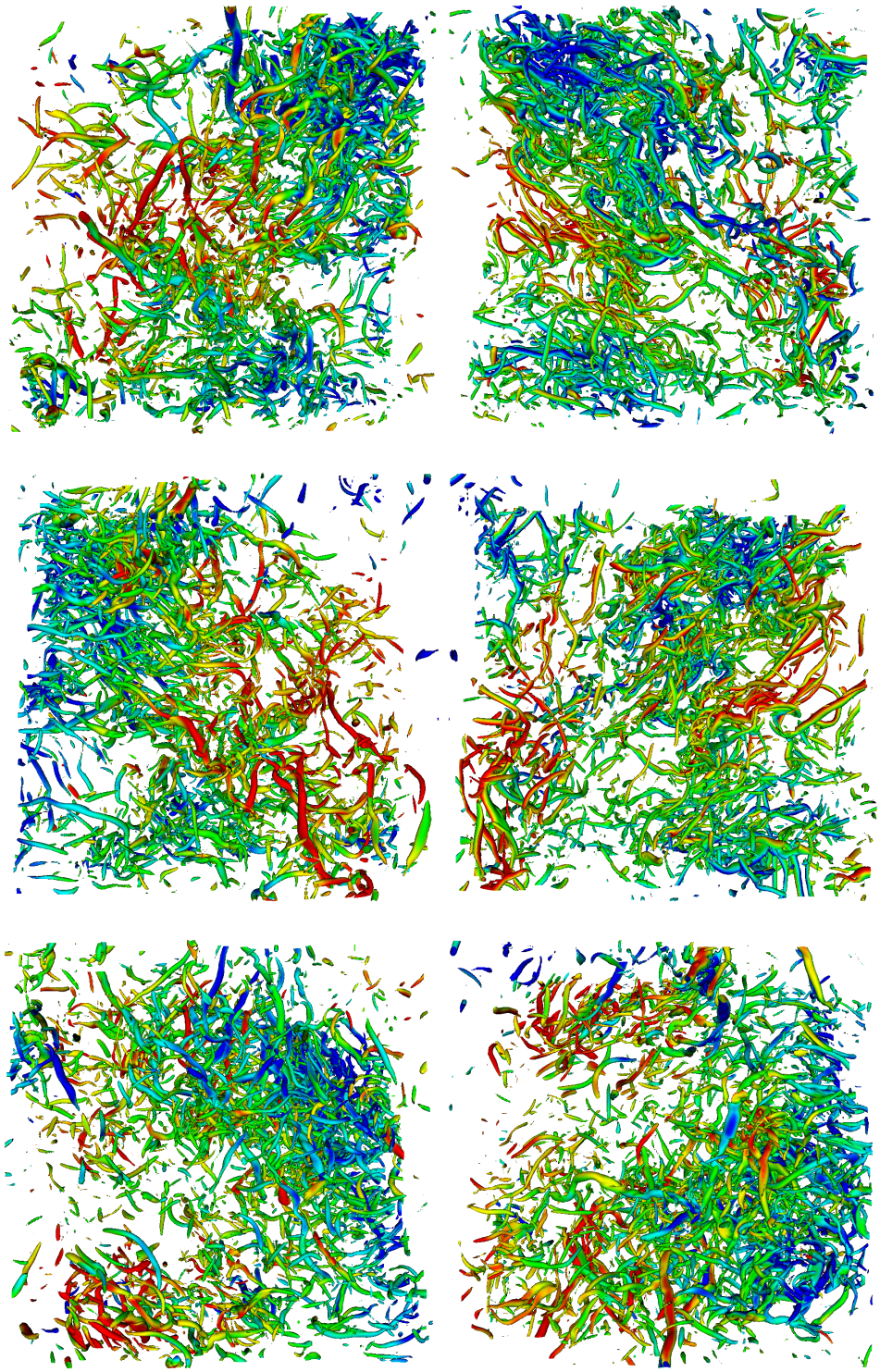

Figure 4.4: Cubes from left to right: Front, Right, Back, Left, Top, Bottom.

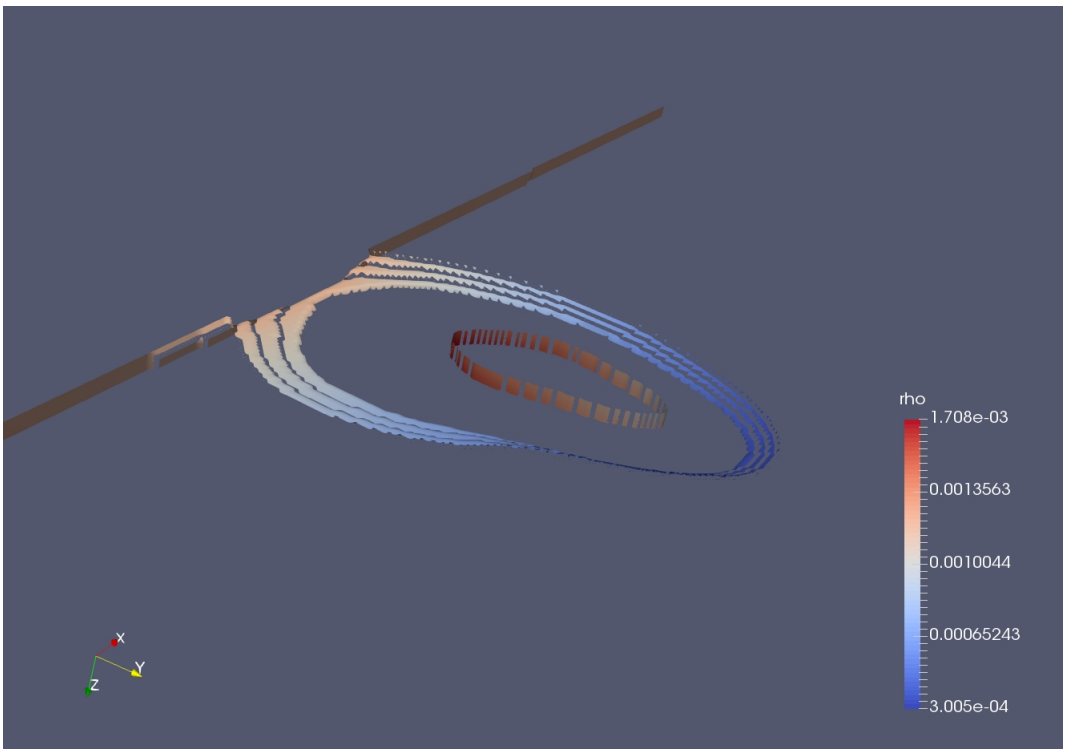

Figure 4.5: A mesh slice created from the simulation of an astroid striking the ocean

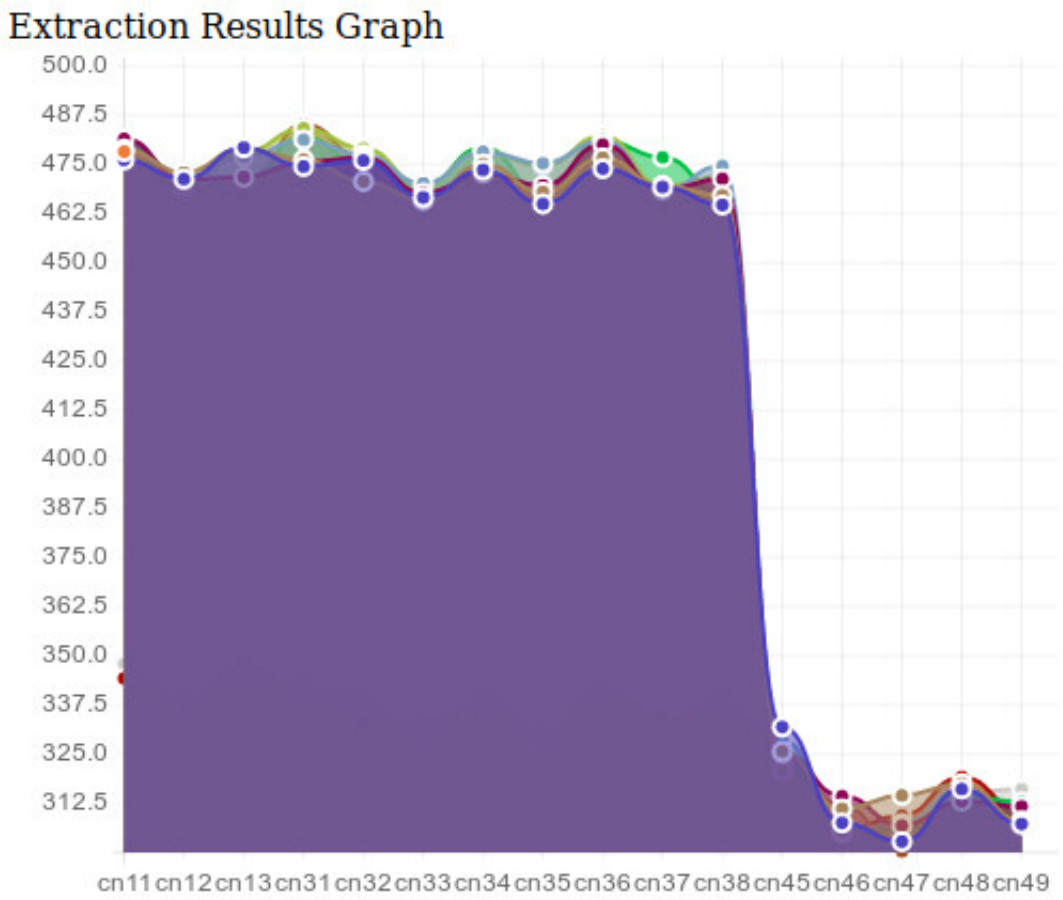

Figure 4.6: Extraction results in total time in seconds for each node

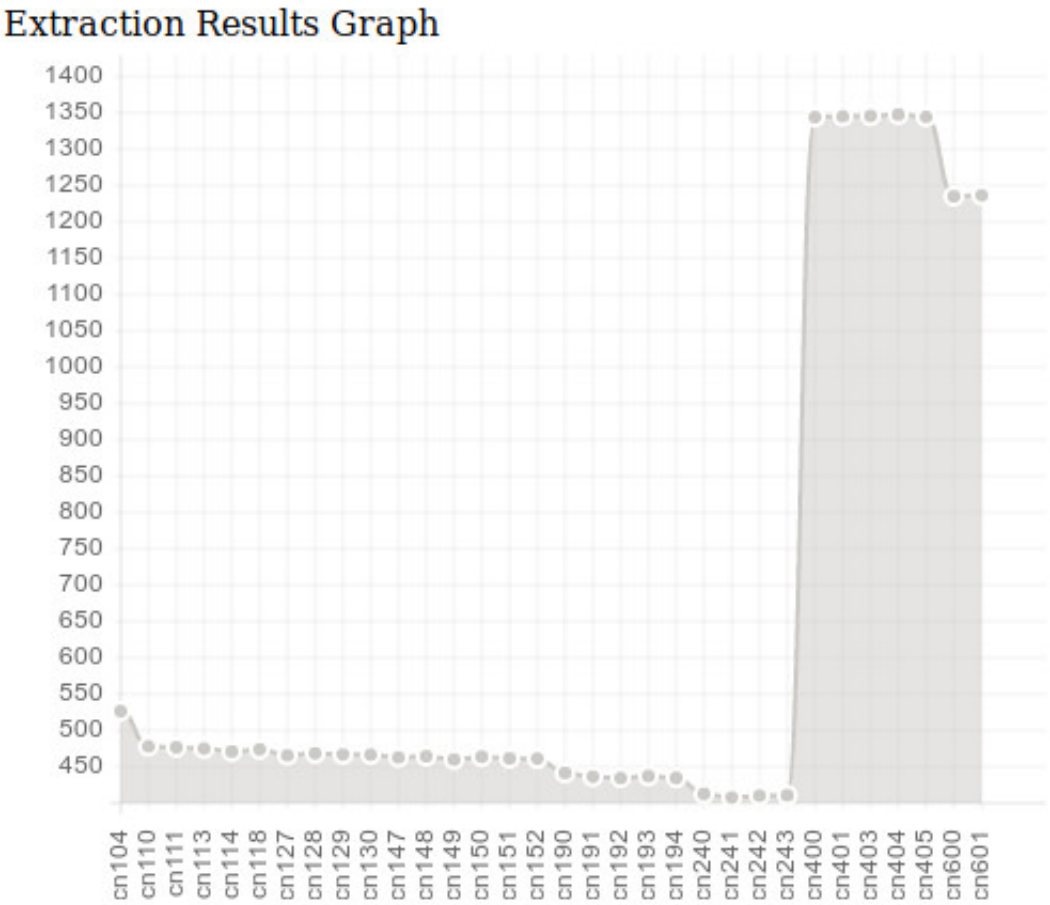

Figure 4.7: Extraction results in total time in seconds for each node

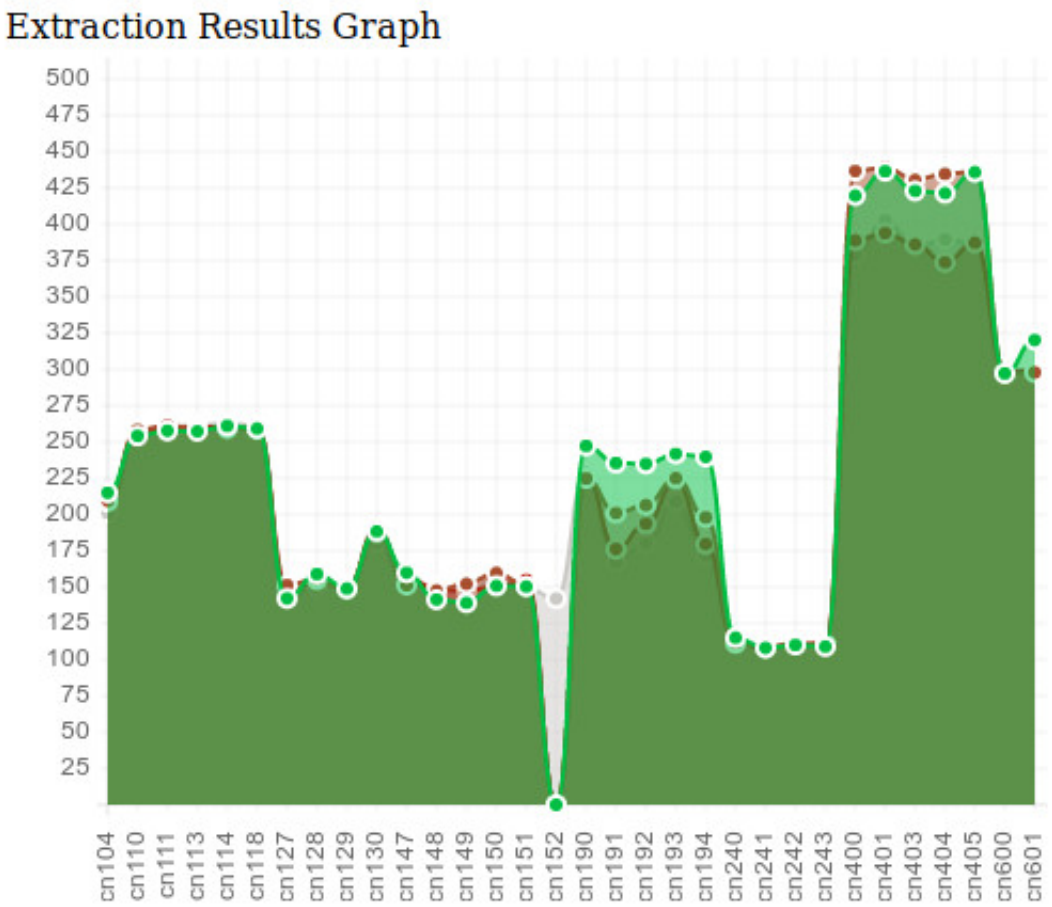

Figure 4.8: zfp compression results in total time in seconds for each node

Chapter 5

Conclusion

This dissertation addresses storage and analysis challenges of numerical simulations as they reach exascale while leveraging next generation simulation architectures. These simulations generate more data than can be saved without interruption due to I/O constraints. We presented Myrcene, a modular distributed parallel extraction system, and ran our extraction algorithms on a mockup of the Trinity Supercomputer burst buffer architecture at Los Alamos National Laboratories.

We began addressing the exascale storage and analysis problem by presenting specific methods of extraction of isotropic turbulence velocity data in high vortical regions. We examined methods of detecting vorticity and tested vorticity magnitude versus Q-criterion. We utilized Q-criterion to detect high vorticity regions and utilized these regions to build a 3D stencil. We expanded the stencil with a defined kernel size and utilized it to extract velocity data from the regions with relatively high Q.

This unique extraction method provided us with a dataset more than an order of magnitude smaller than the original data. The data can be used for scientific analysis where extreme events occur in turbulence, and also for generating visualizations of vorticies within the dataset.

Next we tested and analyzed zfp, a modern lossy compression algorithm designed for scientific data on our datasets. The zfp algorithm outperformed ZLib and LZ4 on speed and final reduction in size. Since zfp is a lossy compressor whereas the other two are not, we further examined how this loss affected the final data. By performing compression of our data with zfp and decompression, we calculated the actual error from the original data, and also visually analyzed the decompressed data. We also utilized the decompressed data to generate visualizations of vorticity by calculating the Q-criterion and contouring the data. The results showed very minor degradation and thus prove it is a viable lossy compressor for visual vorticity analysis.

Finally we built the Myrcene modular distributed parallel extraction system. This system aids the data scientist by providing a modular method of running parallel distributed extraction codes on a burst buffer cluster. We built the previous algorithms into modules and provided additional algorithms for testing and examples. We also built a module that demonstrates the ability to perform snapshot visualization of the extracted data. Any data scientist can build a module to suit their needs and add it to the Myrcene extraction system. The system contains a feature extraction coordinator (FEC) that is web based to allow the data scientist to set up the extraction metadata.

It also contains the modular client where the feature extraction codes are installed. Once the cluster node configuration is entered, the system can be triggered by a simulation to perform extraction on simulation data using a RESTFul call containing the timestep to be extracted. As the extraction modules complete, they report back to the FEC with metadata from the process. This metadata dynamically populates a graph showing extraction times by node. This enables the data scientist to monitor the extraction process throughout the simulation. The system operates seamlessly with or without burst buffers, however burst buffers provide the best performance due to its ability to provide very fast I/O. We utilized the Myrcene system to provide the majority of results throughout this dissertation, since it simplified the complex operations of distributed multiprocessing.

Bibliography

[1] A. Sodani, "Race to exascale: Opportunities and challenges," *IEEE/ACM International Symposium on Microarchitecture, Micro-44*, 2011.

[2] J. Hick, "I/O requirements for exascale," *Open Fabrics Alliance*, 2011. [Online]. Available: http://www.nersc.gov/assets/pubs_presos/OFANERSCExascaleIO.pdf

[3] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12.

[4] N. Hemsoth, "Burst buffers flash exascale potential," *HPC wire, 1 May 2014*, 2014. [Online]. Available: http://www.hpcwire.com/2014/05/01/burst-buffers-flash-exascale-potential/

[5] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns,

S. Chen, A. Szalay, and G. Eyink, "A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence," *Journal of Turbulence*, vol. 9, p. N31, 2008. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/14685240802376389

[6] ACES Team, "Trinity platform introduction and usage model," Los Alamos National Laboratories, number LA-UR-15-26834, 2015.

[7] D. H. Ang, M. Brim, S. Parker, G. Watson, and W. Bland, "Providing a robust tools landscape for coral machines," in *Workshop on Extreme Scale Programming Tools*, 2015.

[8] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–5, lA-UR-pending. [Online]. Available: http://datascience.dsscale.org/wp-content/uploads/sites/3/2016/12/Jitter-FreeCo-ProcessingonaPrototypeExascaleStorageStack.pdf

[9] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012043, 2007.

[10] J. Ahrens, L.-T. Lo, B. Nouanesengsy, J. Patchett, and A. McPherson, "Petascale

visualization: Approaches and initial results," in *Ultrascale Visualization, 2008. UltraVis 2008. Workshop on*, Nov 2008, pp. 24–28.

[11] F. Chen, M. Flatken, A. Basermann, A. Gerndt, J. Hetheringthon, T. Krüger, G. Matura, and R. W. Nash, "Enabling in situ pre- and post-processing for exascale hemodynamic simulations - a co-design study with the sparse geometry Lattice-Boltzmann code HemeLB," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 662–668.

[12] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016, pp. 69:1–69:12.

[13] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–11.

[14] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12.

Washington, DC, USA: IEEE Computer Society, 2012, pp. 945–956. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.89

[15] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2588555.2612185

[16] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2010, pp. 377–388.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[18] Intel, "Hypter threading," *website*, 2009. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J.

Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[20] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[21] D. Zheng, D. Mhembere, V. Lyzinski, J. Vogelstein, C. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication for billion-node graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. DOI:10.1109/TPDS.2016.2618791, 2016.

[22] R. A. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Supercomputing*, 2010.

[23] K. Bürger, M. Treib, R. Westermann, S. Werner, C. C. Lalescu, A. S. Szalay, C. Meneveau, and G. L. Eyink, "Vortices within vortices: hierarchical nature of vortex tubes in turbulence," *Computing Research Repository*, vol. abs/1210.3325, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1210.html#abs-1210-3325

[24] G. Eyink, E. Vishniac, C. Lalescu, H. Aluie, K. Kanov, K. Bürger, R. Burns, C. Meneveau, and A. Szalay, "Flux-freezing breakdown in high-conductivity magnetohydrodynamic turbulence," *Nature*, vol. 497, no. 7450, pp. 466–469, 2013.

[25] K. Kanov, C. Lalescu, and R. Burns, "Efficient evaluation of threshold queries of derived fields in a numerical simulation database," in *Extending Database Technology*, 2015, pp. 301–312.

[26] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[27] K. S. P. K. Yeung, X. M. Zhai, "Extreme events in computational turbulence," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 112, no. 41, pp. 12 633–12 638, 2015.

[28] Y. Dubief and F. Delcayre, "On coherent-vortex identification in turbulence," *Journal of Turbulence*, vol. 1, p. N11, 2000. [Online]. Available: http://dx.doi.org/10.1088/1468-5248/1/1/011

[29] S. Kida and H. Miura, "Identification and analysis of vortical structures," *European Journal of Mechanics - B/Fluids, Volume 17, Issue 4, July-August 1998*, pp. 471–488, 1998.

[30] J. Jeong and F. Hussain, "On the identification of a vortex," *Journal of Fluid Mechanics*, vol. 285, p. 6994, 1995.

[31] H. Tennekes and J. L. Lumley, *A first course in turbulence.* Cambridge (Mass.), London: M.I.T. Press, 1972. [Online]. Available: http://opac.inria.fr/record=b1099467

[32] W. Schroeder, K. Martin, and B. Lorensen, *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 4th Edition.* Kitware, 2006.

[33] R. T. Fielding, "Architectural styles and the design of network-based software architectures," *Dissertation*, 2000.

[34] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003.* Springer-Verlag, 2002, pp. 44–60.

[35] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. [Online]. Available: http://doi.acm.org/10.1145/37401.37422

[36] ——, "Seminal graphics," in *Seminal Graphics.* New York, NY, USA: ACM,

1998, ch. Marching Cubes: A High Resolution 3D Surface Construction Algorithm, pp. 347–353. [Online]. Available: http://doi.acm.org/10.1145/280811.281026

[37] J. M. Patchett, F. J. Samsel, K. C. Tsai, G. R. Gisler, D. H. Rogers, G. D. Abram, and T. L. Turton, "Visualization and analysis of threats from ocean impacts," in *Supercomputing*, 2016.

[38] A. S. Foundation, "Hadoop," *website*, 2002. [Online]. Available: http://hadoop.apache.org

Vita

Stephen Hamilton received a Bachelors degree in Computer Science from the United States Military Academy at West Point in 1998. He attended Auburn University and graduated with a Masters of Science Degree in Software Engineering in 2008. He taught at the United States Military Academy in the Department of Electrical Engineering and Computer Science from 2008-2011 and attained the rank of Assistant Professor. In 2010, he graduated from the United States Military Academy Master Teacher program. He enrolled in the Computer Science Ph.D. program at Johns Hopkins University in 2014.

,